Extempore

The design, implementation and application of a cyber-physical programming language

Andrew Sorensen

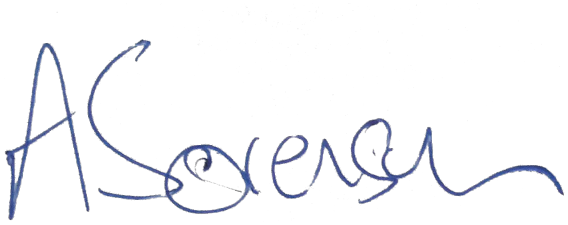
June 21, 2018

A thesis submitted for the degree of

Doctor of Philosophy of

The Australian National University

Except where otherwise indicated, this thesis is my own original work.



Andrew Carl Sorensen

June 21, 2018

Abstract

There is a long history of experimental and exploratory programming supported by systems that expose interaction through a programming language interface. These "live programming" systems enable software developers to create, extend, and modify the behaviour of executing software by changing source code without perceptual breaks for recompilation. These live programming systems have taken many forms, but have generally been limited in their ability to express low-level programming concepts and the generation of efficient native machine code. These shortcomings have limited the effectiveness of live programming in domains that require highly efficient numerical processing and explicit memory management.

The most general questions addressed by this thesis are what a systems language designed for live programming might look like and how such a language might influence the development of live programming in performance sensitive domains requiring real-time support, direct hardware control, or high performance computing. This thesis answers these questions by exploring the design, implementation and application of Extempore, a new *systems* programming language, designed specifically for live interactive programming.

Acknowledgements

I feel incredibly lucky to have had three friends to share my PhD journey with. Without the companionship of Henry, Alistair and Ben, my PhD would have been a far shallower experience.

Firstly, my thanks go to Henry Gardner for his patient supervision and unwavering enthusiasm for the project—after eight years no less! Henry's friendship and good humour have made my PhD experience a joy, from beginning to end.

I am deeply grateful to Alistair Riddell for generously hosting me in Canberra. You made your house my home and I will never forget our conversations long into the night.

I have been incredibly fortunate to have had Ben Swift at ANU throughout my PhD studentship. Initially as a fellow PhD student and increasingly as a collaborator and as a friend.

I would also like to thank Alex McLean, whose "Hacking Perl In Nightclubs" marks the beginning this journey.

Finally to Ali, for all her love and support.

Contents

List of Figures

List of Listings

Overview

1

This thesis explores the cyber-physical system that is the coupling of programmer, program, machine and environment; a relationship between the live programming language (in the broadest sense) and the physical environment. Live programming is particularly sensitive to issues of time and space and requires programming languages, and environments, capable of expressing these ideas effectively. The primary contribution of this thesis is the design, implementation and application of the cyber-physical programming language and environment "Extempore".

1.1. Thesis Plan

The thesis is organised into 6 chapters and 4 appendices. This plan briefly describes the contents of each Chapter, making note of relevant publications and other activities by the author.

1.1.1. Chapter 1. Overview

Chapter One provides a structural overview of the thesis; highlights the contributions and limitations of the research; and relates the thesis chapters to publications and other activities carried out by the author during the PhD.

1.1.2. Chapter 2. Introduction

Chapter Two introduces the central motivation of the thesis — *cyber-physical programming as the live programming of some physical process.* "Liveness" in the context of a live programming practice is discussed, and live-coding, as a musical live programming

practice is introduced. Cyber-physical systems are introduced and the live programming of cyber-physical systems articulated. Throughout the thesis the term "cyber-physical programming" is to be understood as the *live programming of cyber-physical systems*.

Material and ideas from chapter 2. have been previously published in [77], [81] and [80].

1.1.3. Chapter 3. The design of a live language

Chapter Three addresses many of the core concerns for the design and building of live programming languages and environments. Issues of time, concurrency, coupling, reasoning, code specialisation and memory management are discussed. Two broad categories of "half-stack" and "full-stack" are established to help demarcate a boundary between those live programming systems capable of *coordinating* cyber-physical systems on-the-fly (half-stack), and those that are capable of both *coordinating* **and** *building* cyber-physical systems on-the-fly (full-stack).

Material and ideas from chapter 3. have been previously published in [77] and [80].

1.1.4. Chapter 4. The implementation of a live language

Chapter Four provides an overarching description of Extempore's implementation, with a particular focus on the Extempore language (XTLang). XTLang is a new systems level programming language designed specifically for the live programming of cyber-physical systems – for cyber-physical programming. The design, development and application of Extempore (and XTLang), is the central contribution of this thesis.

Material and ideas from chapter 4. have been previously published in [77], [81] and in [80].

1.1.5. Chapter 5. The application of a live language

Chapter Five demonstrates the application of Extempore to a number of substantial case studies. Applying Extempore to nontrivial problems demonstrates cyber-physical programming in-the-wild and the Extempore system's ability to tackle a variety of real-world cases.

The three case studies are:

- A scientific particle-in-cell (PIC) simulation, with material previously published in [85]
- A Computer Music Language with material previously published in [77] and [44]
- A large interactive installation, "The Physics Playroom" with material previously published in[70]

1.1.6. Chapter 6. Conclusions, reflections and future work

Chapter Six concludes with reflections and future work.

1.1.7. Appendix A.

Philip Wadler's expression problem is often used as a barometer of a programming language's range of expression. Appendix A, shows XTLang "solving" the expression problem.

1.1.8. Appendix B.

Appendix B demonstrates an implementation of `List`, taken directly from the Extempore core library. This monadic `List` code provides a reasonably concise demonstration of XTLang's type system.

1.1.9. Appendix C.

Appendix C shows XTLang's core library implementation of a partitioned spectral convolution reverb. A partitioned spectral convolution reverb is a computationally demanding real-time DSP effect. "Live programming" this style of high-performance, real-time problem, on-the-fly, is what Extempore was designed for.

1.1.10. Appendix D.

Appendix D is a website containing an assortment of web resources, including videos of presentations, performances and screencasts, by the author, directly pertaining to this thesis. Many of the thesis code Listings are provided as "live" video screencasts. While the website is not a core component of the thesis, the video documentation helps to convey a level of "liveness" that is difficult to express in text alone. This "dynamic" content, will help to convey a greater sense of Extempore's "liveness" in practice, and provides important documentary evidence of the Extempore artefact.

The URL is repeated here for convenience: `http://extempore.moso.com.au/phd.html`

1.2. Contributions of the thesis

This thesis's contribution has been to identify the priorities that have been implicit in the design of live programming environments and to highlight limiting factors that have restricted the general application of live programming environments to a variety of real-world problems — particularly to those problems to which general purpose systems languages are commonly applied.

The thesis introduces the term "cyber-physical programming" as the live programming

of cyber-physical systems. Cyber-physical programming presents a number of challenges to extant live programming environments and Extempore has been developed to address these challenges.

Extempore, which incorporates the new systems programming language XTLang (a contraction of Extempore language), is the primary contribution of this thesis. Extempore has been pragmatically motivated by a desire to support live programming in real-time physical systems contexts (i.e. cyber-physical programming). Extempore is publicly available `https://github.com/digego/extempore` and has been deployed widely in research, education, arts, sciences and industry.

Three "real-world" case studies, from three separate application domains, help to ground Extempore in practice. These case studies provide a practical and pragmatic grounding for a study of cyber-physical programming practice, and Extempore's ability to support such a practice.

1.2.1. Research Publications

This is a complete list of publications made by the author, as primary or co-author, during the course of the PhD.

- 2010 Onward! (OOPSLA, ACM) [77]
- 2010 International Computer Music Conference [76]
- 2013 Interactive Tabletops and Surfaces (ACM ITS) [70]
- 2013 Visual Languages & Human Centric Computing (VL/HCC) [83]
- 2013 Live Programming (1st International Workshop) [86]
- 2014 Human Factors in Computing Systems (ACM CHI) [84]

- 2014 Computer Music Journal (MIT CMJ) [81]
- 2016 Supercomputing Frontiers and Innovations [85]
- 2017 Onward! (Splash, ACM) [80]

1.2.2. Presentations

During the course of the PhD the author has presented on Extempore, and cyber-physical programming, at over 50 international conferences. Listing only the author's keynote presentations during this period:

- Lambda Jam (Chicago)
- GOTO (Copenhagen, Aarhus)
- OSCON (Portland)
- Pioneers (Vienna)
- CodeMania (Auckland)
- SOSCON (Seoul)
- ICLC (Hamilton, Canada)
- GEECON (Krakow)
- Compose (Melbourne)

1.2.3. Public Performances

The author has performed Extempore live-coding concerts all around the world — from nightclubs in London to palaces in Vienna and concert halls in Copenhagen. Video recordings for some of these "code" concerts can be found in Appendix D.

1.3. Limitations of the thesis

This thesis has been motivated by a pragmatic desire to investigate new physical domains for live programming experiment and practice. To support this exploration, Extempore has developed into a substantial distributed and heterogeneous computational platform, targeting a wide range of systems from single board ARM computers, through to high-performance computing clusters and accelerators. Extempore supports a robust compiler, a stable real-time run-time system, and a substantial core library. Extempore is supported on all commodity operating systems (OSX, Linux, Windows).

A consequence of this "building" and of the pragmatic focus of the thesis is that a thorough theoretical foundation for Extempore, and XTLang more specifically, is not attempted here. Instead, the thesis focuses attention on exploring and applying Extempore, and cyber-physical programming more generally, to problems "in-the-wild".

Addressing problems "in-the-wild" imposes some additional limitations on the thesis. Given a new programming practice (cyber-physical programming) and a new programming language (XTLang), it was decided that a series of end-user studies targeting "real-world" domain problems was unrealistic. Instead, the thesis has adopted a practice based approach, where a number of non-trivial case studies are developed and explored in detail by the author.

There is a danger in practice-based research that the reflections of the researcher, as both tool-maker and tool-user, lack generality. How then to provide the reader with confidence about the thesis's integrity? First, and foremost, Extempore is an artefact that *can be interrogated*. This is a significant, and non-trivial statement; it is common in Computer Science research for the artefact to be of secondary concern to its theoretical foundation and such a focus results in artefacts that are frequently only partial realisations of the theoretical "meat" of the work. These partial artefacts are generally not

practical as vehicles for validation. That is not the case for the work in this thesis.

Second, all of the claims made about Extempore in this thesis can be tested and verified independently, by building and *using* the Extempore software. Extempore is an open-source project hosted on GitHub `https://github.com/digego/extempore` that, at the time of publication, has been forked by 103 people. The project is actively maintained with 4,637 commits made by 32 contributors. Thirdly, this thesis references a substantial amount of additional video documentation `http://extempore.moso.com.au/phd.html` that can be used to verify many of the thesis' claims without the reader having to directly run Extempore. This video documentation is a core component of the thesis' empirical evidence and includes documentation for all of the included case studies. By "building" infrastructure supporting significant explorations across a number of independent domains, the thesis attempts to explore the boundaries of a general purpose programming language for cyber-physical programming.

Introduction

2

> Computer science is an empirical discipline... Each new program that is built is an experiment. It poses a question to nature, and its behavior offers clues to an answer.
>
> Allen Newell (1975)[64, p.114]

2.1. Prelude

Your robotic space probe is 100 million miles away from earth when it manifests a race condition. Luckily for you, your space craft is running Lisp, giving you the opportunity to live (re)program the space robot *without interrupting* the ongoing execution of its critical subsystems. By 5pm the race condition is fixed, the mission is back on track, and you are off home for dinner. Fanciful?

Of the many compelling examples of "live" interactive programming it is hard to go past the story of the NASA spacecraft Deep Space 1. Launched on October the 24th 1998 Deep Space 1 was NASA's first ion propelled spacecraft[63]. When it was over 100 million miles away from Earth a race condition was identified, and rectified, via a Lisp read-eval-print-loop(REPL)[36]. The REPL, which (R)eads a programmer's input, (E)valuates the entered expression, (P)rints any results and then (L)oops back to the read state, allowed Ron Garret, a NASA engineer, to make on-the-fly changes[1] to the space craft's software mid-flight. Ron Garret's experience "live programming" Deep Space 1 is summarized on his website:

> Debugging a program running on a $100M piece of hardware that is 100

[1] As on-the-fly as a 100 million mile transmission time can be!

> million miles away is an interesting experience. Having a read-eval-print loop running on the spacescraft proved invaluable in finding and fixing the problem[28].

This thesis explores the design, implementation and application of a systems language designed to support "live programming" as a first class concern. A primary motivation for this work is to explore the possibilities afforded by an open and ongoing dialogue between the programmer, the program, the machine and the environment.

2.2. Liveness *in* Programming

Live programming[34, 58], live coding[20], interactive programming[65], just-in-time programming[73, 69], cyber-physical programming[77] and on-the-fly programming[96], are phrases used to describe a programming practice that places a strong emphasis on a programmer's ability to manipulate a program's computation via direct source code edits. For such a "live programmer", the program's source code *is* the program's human computer interface, and human computer interaction is, at least in part, mediated through a program's representation.

The activity of programming is intrinsically tied to a program's source representation, be that graphical, textual, or some other form of notation. To be engaged in the activity of programming is to be either *editing*, or *comprehending*, a program's source code. Live edit and live comprehension introduces both program and programming liveness into the *activity* of programming.

Steve Tanimoto, whose 4 (subsequently 6) levels of liveness[88] provided an early theoretical model of programming "liveness" suggests that "In live programming, there is only one phase, at least in principle. The phase involves the program constantly running, even

as various editing events occur"[87, p.31]. This ongoing liveness relationship between the programmer and the program is significant, as it places the human at the centre of the computational system.

Live programming affords liveness relationships for both editing and comprehending a program's *source code*. These are two sides of a bi-directional liveness relationship between the program's source code and the program's computational unfolding; a computational unfolding that often occurs through time, as is the case in a traditional program execution model; but may alternatively occur in space, as a timeless retroactive data structure[65].

What distinguishes a *live edit* from any other source code edit is the direct impact that the change is expected to have upon the program's *ongoing* computational state. All programmer source code edits are "live" in the sense that the programmer is actively engaged in changing the program's representation in real-time. Here the term *live edit* is used more specifically in reference to environments that support some ability to modify a program's ongoing behaviour in direct response to program edits. *Live edits* induce live feedback.

One way in which live feedback can be realized is in the **relationship between edits to a program's representation and its computational unfolding(s)**. At its most extreme this relationship can be presented as a timeless computational expansion where the programmer's edits are made directly to the computational unfolding. Roland Perera presents this computational unfolding as a persistent, reactive data structure[65]. Perera's presentation of "live programming" enables live edits to be directly reflected as alternate computational unfolding's; as future histories. These alternate *future histories* allow a complete computational unfolding to be presented to the programmer as a direct reflection of the program's source representation.

Jonathon Edwards states in regard to his SubText language that "the representation of a program is the same thing as execution"[26, p.505]. Edwards highlights the degree to which this style of live programming is focused inwardly on the improved understanding of the domain of programming itself — an important, and understandable motivation from within the domain of programming to improve its own comprehension.

An alternative path to live feedback can be found in the **relationship between edits to a program's representation and its ongoing effect in the world**. This view of live feedback focuses attention on the task domain rather than the program domain. Edits to the program's source representation are reflected in effective changes in the task domain. In this view of *live edit* the program's source representation need not be a direct, nor indeed even a complete representation of the program's ongoing computational state. Instead the program's source representation serves as an interface for defining procedural action; a loose coupling of a program's source representation and its execution. This is live comprehension, not in the service of a software artefact, but rather in the production of some physical effect, and through that effect some greater comprehension of the physical world.

With regards to the physical world, a program can be considered to be more or less "live" in relation to the natural flow of events that surround it; in relation to the subject, or task domain to which it is applied. A distinction is commonly drawn between "online" and "offline" processing systems, between "real-time" and "non real-time" systems, and between "interactive" and "non-interactive" systems. These are commonly understood liveness relationships in computing based on a systems ability to "keep up" with the world around it. Significantly, in the context of live programming (live edit) the source code at any point in time will only ever provide a partial view of the active run-time system. This is a vision of live programming that highlights code as a transient interface

for intervention rather than a static specification, or a "live" representation.

This thesis focuses on the application of liveness to the external comprehension of the domain *problem*, rather than the internal comprehension of the *program* itself. From the programmer's perspective, a focus on liveness in the world rather than liveness in a computational unravelling. More concretely, the focus here is not primarily with the live inspection and interrogation *of a computation*, but instead with the live inspection and interrogation *of a physical process*. Code as a transient interface for interacting with the world.

2.3. Live Programming

By focusing attention on the relationship between edits to a program's representation and its ongoing effect in the world the attention is focused on a coupling of programmer, program, machine and environment; a liveness relationship between the human programmer and the task domain. Live programming a system that keeps up with the world around it integrates the human programmer into an ongoing flow of natural events; a flow of natural events that the human programmer is able to sense and act (procedurally) upon. In the context of live programming, a natural flow of events that must ultimately bear some relevance to the human programmer — to the programmer's sense perceptions. Ultimately, if the ongoing environmental effect of the computation bears no relevance to the human programmer then there is little point in "live programming" as a programming practice.

A focus on task effectiveness prescribes that algorithmic description and evaluation occur within a temporal quantum that is appropriate to a given task domain. As Richard Potter suggests:

> ... the goal of [live programming] is to allow users to profit from their task-time algorithmic insights by programming. Instead of automating with software what was carefully designed and implemented much earlier, the user recognises an algorithm and then creates the software to take advantage of it just before it is needed, hence implementing it just in time.[69]

Potter's emphasis on the "user" as a *spontaneous* algorithm creator is fundamental to live programming and distinguishes it from other more traditional design-centred software engineering practices. Live programming promotes the idea that computer programming can be a method of production whose artefact is *not primarily a software product.* Once the programmer has manifest a graphic on a computer screen, a sound wave, moving a robotic arm, a blinking LED etc., any source code written, executed or modified during the process of creation may be modified in anticipation of future action, or potentially discarded entirely. During a live programming session source code that was valid at the start of the session is very likely to be re-appropriated for a different purpose by the end. Source code will expand, contract and morph during the course of a session. Significantly the source code at any point in time will only provide a partial view of the active run-time system. Live programming is ephemeral.

Live programming aligns with an iterative and incremental development cycle common to Agile software development methodologies[8]. Like Agile methods, live programming advocates a style of negotiated development between programmer and computational work. Agile methods acknowledge the limitations of formal specification in the development of real-world projects and attempt to systematise the development cycle to accommodate unknown, or poorly known, requirements[8]. Where live programming diverges from Agile thinking is that live programming is fundamentally more transient in nature. Not only is development a negotiation, it is also ephemeral. Live programming is often

about experimentation rather than manufacturing. In the words of Erik Sandewall:

> The average Lisp user writes a program as a programming experiment ... in order to develop the understanding of some task, rather than in expectation of production use of the program.[75, p.39].

2.4. Live "Systems" Programming

In the context of live programming what it means for a program to be relevant to the world around it, and what is required for it to keep up, is ultimately a subjective qualification made by the live programmer himself or herself. Live programming is a human centred activity and any assessment of "liveness" will ultimately be a personal assessment by a given live programmer. Nevertheless, certain liveness criteria can be assumed in relation to human perception and the physical processes being engaged with.

There is "liveness" in a Unix shell session, where commands such as `ls`, `cd`, `pwd`, `mv` etc. form an ongoing dialogue between the user and the operating system. A shell session establishes a style of call and response – single shot executions of small programs designed to provide feedback about the *current and ongoing state* of the operating system. A shell session is a dialogue with past queries and responses helping the user to build a mental model about the state of the system over time. Unix shell expressions such as `find . -name \*.cpp -exec grep "TODO" {} \;` can be reasonably considered *programming*, and Unix shells (bash, csh, tcsh, zsh etc.) are live programming environments.

The one-shot nature of these shell examples, with each command issued waiting on a singular response, might more commonly be regarded as *interactive programming* rather than *live programming*. In this thesis these terms are considered to lie on a continuum of liveness. What makes them suitably live or appropriately live is in relation to human per-

ception and the domain of interest. For interrogating a file system, shell commands seem to provide an appropriately "live" dialogue. Furthermore, there is clearly a sophisticated and very "live" real-time system underlying these shell examples – the operating system. Moreover live edits can run ongoing interventions and queries against this real-time operating system, which may be purposed to provide continuous feedback.

For many interactive dialogues the rate of feedback supported by a shell script, or a Lisp REPL is more than adequate. However, there are many task domains which require considerably higher rates of feedback, (often with considerably more complex calculations) than a standard Unix script is able to support. For computationally demanding, low-level and real-time tasks the Unix programmer invariably turns to C.

There are two primary reasons that C is such a popular systems language for task domains requiring high rates of feedback in computationally demanding domains. First, as the lingua franca of operating systems the lowest level Application Programming Interface (API) available on most Operating Systems is a C interface. For this reason, C invariably provides the most extensive interface for directly programming the operating system. Secondly, the C language's close relationship to the physical machine has often been described (in antipathy as often as in reverence!) as "a portable assembly language"[9]. The most important quality bestowed on C by these two factors is great flexibility, enabling C to be effective in many different domains where performance matters.

Unfortunately for the live programmer, while C is an excellent candidate for *live feedback*, C is not well equipped for *live edits*. While there are interactive interpreters for the C language, and C debuggers provide some basic live edit capabilities, C and the C language's infrastructure more broadly, were never designed, nor intended, to support live edit. Instead, C is the quintessential batch-compile language where a static program

text is compiled into a static binary image, removed from its source code (debugging aside).

Shell scripting provides C programmers with a level of *live programmability* by making their C programs available in end-user shell sessions. However, this interaction only goes so deep, with a significant divide separating interactivity in the Unix shell, from interactivity at the C source code level. C provides a level of run-time performance and access to the physical machine that Unix shell scripting environments do not. C provides the ability to program across the full application stack, while shell scripting is limited to only part of the application stack. As live interaction is limited to the shell it can be argued that this also limits a Unix programmer's ability to **live program** to only "half-stack" live programming – half in the non-literal sense of something less than full. Unix programmers can live edit some, but not all of their environment.

2.5. Live programming "Full-Stack"

The general ideal of "full-stack" live programming can be traced back at least as far as Smalltalk. Alan Kay envisaged a future where computer users would be computer programmers - computer programmers empowered with the "magic" to conjure new materials from the machine as if from a crucible[2]. Central to this ideal was not simply the Smalltalk programming language, but more profoundly the Smalltalk operating environment - Smalltalk *as* operating system. "Programming Language as operating system" was also at the core of Lisp Machine Operating Systems, such as Genera[95].

Both Alto Smalltalk and Lisp Machines provided a "full-stack" interactive development experience. "Full-stack", in the sense that both of these environments provided the ability to make substantive changes to the end-user's real-time run-time operating system in a live way. Classic "full-stack" live programming examples from both the Smalltalk and

Lisp Machines communities include implementing and testing low-level network protocols, graphical interface stacks and operating system scheduling[14][45] – all programmed interactively and on-the-fly.

The two most significant factors that distinguish a "full-stack" experience from a "half-stack" experience are; *accessibility* to low-level operating system features; and the *performance* required to adequately express these low-level operating system features. Accessibility is required to *re-implement* the network stack on-the-fly, and good performance is required to make those changes *effective*.

In reference to Smalltalk, Daniel Ingalls highlights "accessibility of the kernel" along with "incremental compilation" as providing "high productivity, deriving from the elimination of conventional loading and system generation cycles"[45, p.22].

> Smalltalk's incremental compilation and accessibility of kernel code encourages you to make the change while the system is running, a bit like performing an appendectomy on yourself.[45, p.22]

Ingalls goes on to highlight that *speed*, as well as *accessibility* is required for true "full-stack" operation.

> Speed comes into play here, because if the kernel is not fast enough it will not support certain functions being implemented at a higher level.[45, p.22]

Smalltalk attempts to be accessible by utilizing a relatively minimal microcode "kernel" (which might be C or ASM depending on implementation), above which the Smalltalk OS is written in Smalltalk. By minimizing the footprint of the "inaccessible kernel", Smalltalk makes most of the operating system available for on-the-fly Smalltalk code modification, and hence for live programming "full-stack"[2]. Lisp Machines go one step

[2]The Squeak Smalltalk implementation supports a direct translation from a limited subset of Smalltalk into C. This allows the Smalltalk "kernel" to be written in Smalltalk and compiled directly to C [42].

further by tailoring the machine's architecture to the execution of Lisp (in this sense there is no layer too deep for live programming on a Lisp Machine[14]).

Yet, despite the substantial influence that these systems have had on the landscape of computing, and computing folklore, they remain a relatively obscure blip in the history of *programming practice*. It is difficult to fully appreciate why this might be the case, but three issues seem likely. Firstly, these systems were expensive and not widespread. Secondly, both the original Smalltalk systems and Lisp Machine systems, were plagued throughout their lifetime with complaints of poor overall system performance. Finally, they were worlds unto themselves, and not easily integrated into or with *other* computing platforms.

Meanwhile, Unix and the C programming language have thrived. What Unix and C made possible, was more effective utilization of the available hardware and a more open development platform. What was sacrificed was an integrated, and interactive development experience, that allowed people to re-program systems full-stack and on-the-fly, right down to the hardware.

2.6. Live programming as a Performance

At the dawn of the new millennium a number of digital artists were beginning to explore live programming in musical performance, with the program's source code projected for all the audience to see. These early live-coding[3] pioneers[20] integrated programming languages into their sound processing environments. Along the way they developed several ad-hoc half-stack live-coding environments that coupled high-level "scripting" languages onto low-level "native" audio processing frameworks[20, 96, 78].

[3] Live-coding is a term used in this thesis to denote a live programming sub-domain centred around musical performance.

With these new "live-coding environments" in hand, these software artists began exploring the notion of sound as a direct representation of a program's state. Live-coders embraced live programming, not only as a means to explore an artistic design space, but also as a performance practice, an unveiling of the often hidden world of a program's source code. "Show us your screens" was an early "demand" from the TopLap manifesto[92], a reactive stance against both the obscurantism of "Laptop Music" (is that guy actually doing anything, or just pressing play in iTunes?) and a society that is completely beholden to software, and yet is largely ignorant of the "source code" from which it is derived[54].

Figure 2.1.: The author live-coding music with two robotic pianos in the Hofburg Palace Vienna.

Over the intervening years, the community of live-coding practitioners has grown in size and has increasingly grown away from its largely "academic computer music" roots. Live-coders, including the author, are regularly invited to demonstrate live-coding practice at international software developer conferences. From the personal experience of the author,

what is interesting about this popularity is not that software developers are surprised that code can be used to generate music (although that is surprising), but that they are often surprised that this style of on-the-fly programming can be accomplished at all — at a technical level.

As highlighted earlier, live programming is not a new concept and its roots stretch back to the birth of computing. The ability to change and modify a program's behaviour in situ are common-place in many industries. The games industry makes extensive use of live "in game" scripting, and has been doing so for at least as long as musical live-coding has been commonly practiced. Yet there does seem to be *something* about live-coding music that is different, somehow separate and distinct from the live programming of the past. While it is difficult to point to specific factors that distinguish live-coding, it is possibly found in the "instant" — *the now*.

The meaning of an instant in time is a problem that has long preoccupied philosophers and psychologists. In signal processing, engineers also give much thought to appropriately modelling an instant in time from a stream of data. The treatment of an instant in music is recognized by musicians as being intrinsically tied to notions of performance, virtuosity and improvisation.

What appears to be unique to musical live-coding, is that live programming is intrinsic. Live-coding was born *of* live programming, and does not exist in any form separate from it. Games programmers may engage in "live" experiments by live programming game state changes, but they do not write entirely new games on-the-fly, in someone's lounge room, just ahead of the player. A live-coder, on the other hand, is programming a bespoke piece of music, in someone's auditorium, just ahead of the audience. Without live programming there is no concept of live-coding.

There are then both social, and technical, relationships to time and space, or perhaps

more accurately place[25, p.89], that arguably appear more strongly in live-coding practice, than in wider live programming practices. Live-coders must listen to the sounds that their algorithms are producing in real-time. They must be reactive to those sounds, supporting the ability to rapidly respond to changes in acoustic conditions that may support, or negate their efforts. They commonly perform with others, both acoustic and digital musicians, and must naturally fit within a collaborative and physical acoustic whole. They perform, they (generally!) care about their audience, and they make every effort to produce something of value, unique to a particular time and place. They do all this within an extremely demanding technical environment that embeds the live-coder as an integral component in a complex cyber-physical system.

2.7. Cyber-physical systems

It can be argued that the dominant computing paradigm of the modern era is neither "number crunching", nor information processing; although both of these are still very important. Instead, the majority of new microprocessors are being embedded in an assortment of manufactured goods, where their primary purpose is to regulate the behaviour of highly dynamic systems. As these systems become increasingly interconnected (to each other and to the physical world around them) and their behaviours more sophisticated, their software systems become increasingly complex. These interconnected physical computing systems are known as cyber-physical systems.

Cyber-physical systems are integrations of computation with physical processes[51]. While they are derived from embedded computing systems, cyber-physical systems are marked by a particular set of requirements that make them unique in their own right. Rajeev Alur cites these five key concerns as central features of cyber-physical systems[4].

Reactive A reactive system's interaction with its environment is ongoing. Where a classical computations correctness can be captured with a mathematical function mapping inputs to outputs, a reactive system's correctness is naturally described by a sequence of *observed* inputs and outputs and their correspondence to *acceptable* behaviours.

Concurrent Concurrent computation involves multiple parallel processes of execution communicating information in the service of a coordinated goal. For sequential computation the Turing machine provides a canonical model of computation. No such agreed *formal model* exists for concurrent computation. Instead there are many competing formal models for concurrent computation. Broadly these can be broken down into synchronous and asynchronous modes. Cyber-physical systems make extensive use of both synchronous and asynchronous models.

Feedback A control system interacts with the physical environment via a feedback from sensor readings to actuator adjustments and round again in a continuous feedback loop. Control systems, and their integration with discrete digital computing systems, are one central concern of cyber-physical systems.

Real-time "Programming languages and the supporting infrastructure of operating systems and processor architectures, typically do not support an explicit notion of real-time"[4, p.4]. And yet a notion of continuous real time is required to support the control systems central to cyber-physical systems.

Safety-critical It is often the case that cyber-physical systems have a strong safety component. Fly-by-wire systems for aircraft flight control are safety critical cyber-physical systems.

This is a challenging list of requirements for both software developers and programming

language designers: reactive programming, concurrent programming, real-time programming, embedded-systems programming and security programming. On top of this set of programming requirements the underlying motivation of this thesis is to know whether it is feasible, to not simply *program* cyber-physical systems, but more radically to *live program* cyber-physical systems — creating cyber-physical systems that intrinsically incorporate a human programmer within-the-loop.

2.8. Cyber-physical programming

The term "cyber-physical programming" defines a *live programming practice* where the target domain is a *cyber-physical system*. A premise of this thesis is that there are domains for which the addition of a human agent (the programmer) is either necessary, or highly advantageous, for success in a particular task domain.

Cyber-physical programming provides a model for human supervised systems. Cyber-physical programming diverges from fully autonomous systems by integrating a human programmer in-the-loop. Cyber-physical programming also diverges from most semi-autonomous systems by shifting the focus of human interaction away from direct manipulation and towards meta information processing — the directed orchestration and ongoing maintenance of many concurrent information processing sources. From an information perspective the cyber-physical programmer fulfils a meta role, selectively influencing a continuous stream of real-time information. This information is sieved through the system by the careful manipulation of higher level processes. The cyber-physical programmer does not directly manipulate the flow of information but instead *programs* a network of cyber-physical agents to intervene on the programmers behalf. This places the programmer *within-the-loop* but also *above-the-loop*.

While the cyber-physical programmer's role is usually second order, it is not strictly

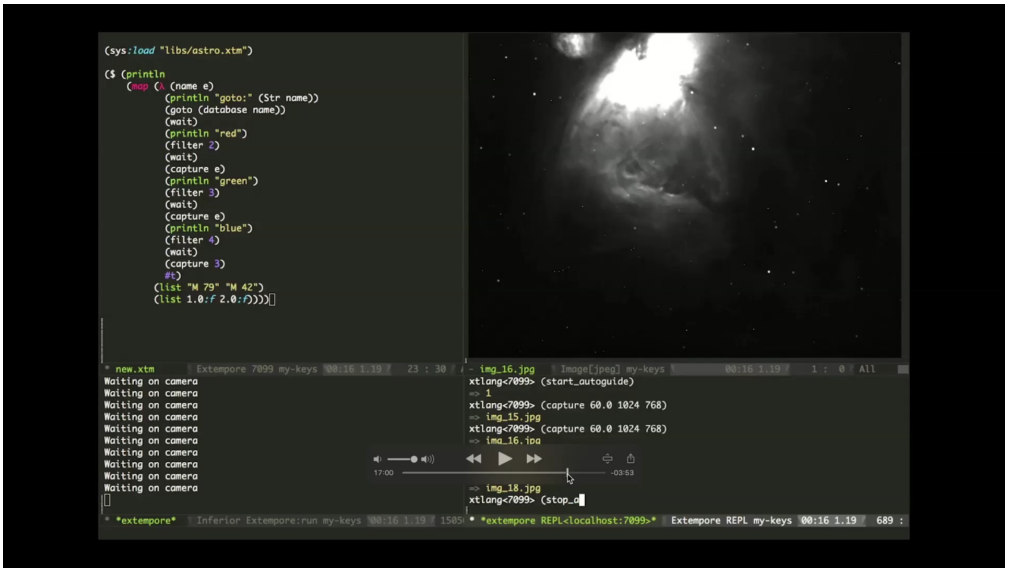


Figure 2.2.: Cyber-physical programming a robotic telescope with Extempore.

second order. The human agent may also provide a primary duty as an additional, and far more complex, sensory organ and information analyser; an *active* and *direct* participant in the behaviour of the system. What cyber-physical programming attempts to provide is a balance between automation and control *from a human perspective*.

What distinguishes cyber-physical programming from a more traditional programming practice are the real-time constraints imposed by the cyber-physical system being developed/controlled. These real-time constraints affect the overall performance of both the programmer and the underlying cyber-physical system. By making software development a real-time activity, software itself must become run-time malleable. It must become amenable to change in real-time at run-time. As cyber-physical programming targets the *building*, as well as the *orchestration*, of cyber-physical systems, the tools of the cyber-physical programmer must adequately address Rajeev Alur's five key concerns for cyber-physical systems, as outlined in the previous section. Significantly, cyber-physical programming tools must address these concerns whilst supporting *live edit*.

2.9. Musical live-coding as cyber-physical programming

Live-coding is a form of live programming that is highly concurrent, reactive, real-time and involves significant feedback between the programmer and the sonic environment. Whilst musical live-coding may be considered to be cyber-physical in nature, in the context of this thesis, the term "cyber-physical programming" is used more specifically in relation to systems that are not only capable of *orchestrating* an audio signal processing system on-the-fly, but also of *building* and/or *modifying* that audio signal processing system on-the-fly.

Live-coding converges with cyber-physical programming when the target of on-the-fly development is an audio signal processing system — a *firm* real-time system[17]. Although live-coding systems are not *hard* real-time systems, a failure to meet all scheduled deadlines during a musical live-coding concert, in front of a live audience, is extremely detrimental to the overall performance, and highly embarrassing for the performer. In extreme cases, such as high decibel concerts, it is even possible to cause audience members physical (aural) damage. Missed deadlines in the underlying audio stream, measured in microseconds, represent a failure of the system, not simply a quality-of-service (QOS) issue. Time, becomes a correctness issue, rather than a QOS issue for the live-coder[50].

A core concern for cyber-physical programming, and of live-coding practice, is the programmer's ability to orchestrate a concurrent set of ongoing cyber-physical processes. Through the writing, editing and execution, of a program's source code, the programmer builds and modifies the behaviour of computational processes within the temporal bounds of the performance; within the temporal bounds of the *task domain*. Success is predicated on the programmer's ability to build, execute, and modify these processes in a *timely* manner. For a musical concert, a timely performance would meet the audience's expectations of a meaningful musical form based on a common set of cultural norms[60].

The piece would be expected to conclude within a certain time period; might include an introduction of material, followed by a development and conclude with a recapitulation of ideas; would introduce novelties into the form at regular intervals etc.. In short, the programmer would be expected to change the program regularly in order to successfully construct a *meaningful* performance. Where *meaning* within this context is a cultural phenomenon realised in a *social place*, not merely within a *physical space*[25].

There are then two central temporal concerns demanded of a cyber-physical programming system. There is a requirement that the cyber-physical programming system provide a language interface that supports the programmer's desire to orchestrate computational processes in real-time (higher-level computational support). Whilst at the same time, this same system must be capable of realising the programmer's source code (intentions) into real-time computational processes capable of meeting the physical demands of the task domain (low-level computational support). A general purpose programming environment for cyber-physical programming must support both the timely orchestration of computational processes *through a programming language interface*, as well as the strict firm real-time computational demands necessary to interface correctly to the physical environment — in the case of live-coding, to the sonic environment.

In order to realise this high level of performance and flexibility on modern multi-core architectures, low-level access to the machine is as important today as it has ever been. High performance audio-visual code is highly parallel (multi-core), vectorized (SIMD), memory aware (NUMA), and often makes extensive use of extended CPU instructions, on heterogeneous architectures (x86,ARM,PHI,GPU,DSP).

Live-coding presents a challenging scenario for cyber-physical programming systems developers. Not only must the system provide the high numerical performance and physical (sensor/actuator) connectivity required by modern real-time audio-visual ap-

plications, it must also manage these real-time demands whilst providing the human programmer with on-the-fly access to the tool-chain — compiler, linker, debugger.

Ultimately the *tools* of cyber-physical programming must meet the real-time constraints demanded by *firm* real-time systems, not just in the domain of audio and motion graphics, but across a broad spectrum of domains from sensor networks, to robotics, to machine-learning and data mining, to high performance computing and interactive display technologies. This is the broader context of live-coding *as* cyber-physical programming.

The design of a live language

3

The following sections discuss the design of a cyber-physical programming environment. The discussion is broken into two halves. The first half investigates *half-stack* live programming environments. The second half then builds from *half-stack* live programming design to more fully incorporate the idea of a *full-stack* cyber-physical programming environment.

Throughout this thesis full-stack refers to a hierarchy of software abstractions reaching down to, and into, the operating system kernel. Simply speaking, a full-stack programming environment is any programming environment that allows the programmer to work at all levels of the software stack; in principle allowing the Operating System's kernel itself to be written or updated on-the-fly. By contrast, a half-stack environment is any live programming environment that falls, for one reason or another, short of this expectation.

3.1. Half-stack live-coding

The primary motivation for building a half-stack live programming environment is to introduce a degree of *programmability* to some *extant* real-time system. To augment the functionality of some existing real-time system through the introduction of a programming language *interface*. In principle to provide a more powerful/flexible/abstract/expressive interface for the human control of the underlying real-time system than would otherwise be available for run-time interaction.

In the music live-coding community, the primary motivation has been to provide greater expressive control of real-time digital audio signal processing systems. These real-time digital audio signal processing systems and computer graphics systems are

almost exclusively written in either C or C++, with a few prominent examples including CoreAudio, ASIO, DirectX, OpenGL, PortAudio, Jack, SuperCollider Server, VST, AudioUnits, Open Frameworks, and Dirt. C/C++ provides the performance and deterministic run-time behaviour that make it not only the lingua franca of operating systems development, but also the lingua franca of real-time systems development, including firm real-time multi-media systems. Unfortunately C and C++ offer limited out-of-box support for *live edit* making them generally unsuitable as live programming interfaces.

It is possible to introduce *live edit* post hoc to these extant real-time systems by overlaying a higher-level orchestration language. While the orchestration language will not generally support the ability to change the internal implementation of the real-time C/C++ library, it will provide the ability to orchestrate the C/C++ library at its public interface – exactly as the Unix shell is able to orchestrate the behaviour of the operating systems at its public interface.

A standard approach to developing audio-visual live-coding environments has been to take an existing, managed higher-level programming language, such as Ruby(Sonic Pi[1]), JavaScript(Gibber[71]), Scheme(Impromptu[78], Fluxus[30]), Clojure(Overtone[1]), and to couple that managed language environment to a native C/C++ code framework.

The same approach is common in other live programming environments; computer game engines being the most prominent example. Computer game engines commonly use managed languages (particularly common are JavaScript and C#) as higher-level "scripting languages" coupled to low-level native C/C++ real-time game engines. The Unity environment provides one such example[93], although most game engines support some form of higher-level end-user scripting over a real-time C/C++ engine. There is also a significant history of visual languages that take a similar approach by providing higher-level flowchart definitions for the control of low-level native real-time C/C++

frameworks[39].

What these environments share with musical live-coding environments is the coupling of a managed high-level language with a real-time native run-time system. A native run-time system written in a low-level systems programming language distinct, and ultimately removed, from the coupled high-level live programming language interface.

The following discussion of half-stack live programming languages breaks into important qualitative dimensions: of (i) high-level languages (ii) coupling (iii) concurrency (iv) time and (v) latency.

3.1.1. Higher-level Languages

Higher-level languages are often associated with words like interpreted, dynamic and managed. A (very) short list of higher-level languages would include Perl, Ruby, Python, Lisp (Scheme and Clojure), JavaScript, and Smalltalk as well as the domain specific audio live-coding language SuperCollider (sclang). These languages all share the common themes of being interpreted (either direct or byte-code interpreted), dynamic (late binding and dynamically typed), and memory managed (garbage collected or reference counted) [1]. These high-level "dynamic" languages share a number of features that make them particularly well suited to live programming.

1. Dynamic languages generally support late binding, as opposed to the early binding common to systems languages such as Ada, C/C++ and Fortran. Dynamic binding provides the ability to look-up the target of a symbol at run-time, introducing a level of indirection into the control flow. A useful consequence of this indirection is that late binding usually makes it straightforward to *redirect* a symbol's target

[1] While virtually all of these languages have had native code compilers/JITs written for them at some point (SBCL, Iron Python, Rubinius, Chez Scheme, V8, SpiderMonkey ...), their original design was predicated on a model of interpretation

at run-time, a feature common to most dynamic languages.

2. Dynamic languages are generally interpreted languages, which require run-time *interpretation* to translate program statements into appropriate calls to pre-compiled *native code routines*. Taken as a whole, these *native code routines* are usually named "interpreter", or "virtual machine". Due to the inherent run-time nature of this interpretation process, which takes program statements and dynamically translates them on-the-fly to target pre-compiled executable routines, dynamic languages have a natural run-time relationship between source code and execution. The "interpretation unit" is also generally more fine-grained than batch compilation - often down to the level of a single expression.

3. The run-time interpreter required by dynamic languages (as described by point 2.), necessitates a more substantial run-time system than is generally required for systems languages. One positive consequence of this necessary run-time system, is that this additional run-time tooling often makes it straightforward to incorporate direct programmer intervention into the system. A typical extension offered by interpreted languages is an interactive shell, or REPL, which integrates the parser (R-read) and interpreter (E-eval) into a top level interaction (P-print) loop (L-loop).

3.1.2. Language and framework coupling

When coupling a higher-level "dynamic" language, to a lower-level "systems" framework, a decision must be made about how tight or loose to make that coupling. From a high-level perspective a decision must be made between intra versus inter process communication. Intra-process communication within a shared address space (e.g. within a single Unix process) and inter-process communication between separate memory address spaces (e.g.

between separate Unix processes).

Intra-process communication requires that the language and the framework share the same memory address space. Generally speaking this means that the language and the framework must be coupled within the same process [2]. Intra-process communication has the advantage of supporting direct calls from the language to the framework, generally mediated through a Foreign Function Interface (FFI). FFIs can be developed to bridge between any two languages Application Binary Interfaces (ABI), although by far the most common FFI bridge is to the C language.

Most higher-level languages ship with some form of C-ABI FFI; and despite the increasing adoption of higher-level languages, the C-ABI has remained a common target for language agnostic native systems frameworks. A well written C library can be readily adopted by any number of higher level languages.

Intra-process communication, through an FFI, is often the most flexible and efficient vehicle for coupling systems [3] from both a machine architecture and human productivity perspective. The ability to interface directly to a native framework, on the framework's own terms (i.e. using the frameworks ABI) supports communication never anticipated by the frameworks (or programming languages) authors. Although the C-ABI has no definitive standard, C's dominance in operating system development ensures that, in practice, there are well defined platform ABI's for C. As a result, C-FFI's are common in practice, and C libraries are a popular option for cross language frameworks. Significantly intra-process communication through a C-FFI makes it possible to interface with C libraries that were *never intended* to be coupled to a higher-level *live edit* language. A significant advantage for tight coupling is that it can make possible the use of an extant

[2] Process in the traditional Unix context

[3] Although certainly not always. Poorly designed FFI's can very often make "direct" calls inefficient (from a code optimization perspective), and extremely tedious to work with.

code base that was never intended to support live programming.

This style of intra-process communication has been adopted by many live-coding environments including ChucK[96], Impromptu[78] and Fluxus[30]. Being both efficient and flexible, intra-process communication offers an attractive option for integrating live-coding languages with extant low-level audio-visual frameworks - OpenGL, STK, CoreAudio, QuickTime, DirectX, ALSA, JACK, PortAudio, and PortMidi to name a few.

There is however, a serious disadvantage with the tight coupling of language with framework; the tight coupling of failure! Any catastrophic failure in any of the tightly coupled components, either language of framework, results in failure across the entire system. This becomes increasingly significant as the number of coupled systems increases (i.e. as more frameworks are supported by the live-coding system).

An alternative approach to the tightly coupled intra-process FFI approach is a more loosely coupled inter-process communication (IPC,Actors/Message Passing). Inter-process communication supports the idea of coupled components each operating within their own private process address space. Inter-process communication has two significant advantages over the tightly coupled intra-process approach.

Firstly, it is safer to run the language, and the coupled frameworks, in their own memory address spaces. Any catastrophic failure in one process space is isolated by the operating system, potentially at least, from the other coupled processes. This isolation helps to restrict the overall damage that a single rogue component can cause.

Secondly by isolating the language interface from the coupled frameworks, it makes it far simpler to couple incompatible ABIs. Language "X", with an interpreter written in Standard ML, can be coupled considerably more simply with a framework, written in Java for example, when each is able to run in its own process address space. A loosly coupled inter-process communication approach removes the necessity for an FFI.

An added benefit of the inter-process model, is that *distributed* communication is very often achieved "for free". This, due to the fact that many inter-process communication mechanisms also support inter-node/networked communication.

In the live-coding community the most well known inter-process example is SuperCollider (version 3)[57]. SuperCollider provides an OpenSoundControl[98] (TCP/UDP) connection between the framework (SuperCollider server) and the language (SuperCollider language - SCLang). The SuperCollider communications protocol is extremely rich, providing a level of expression usually associated with a more tightly coupled approach. This is not surprising as versions 1 and 2 of SuperCollider were designed as tightly coupled systems; that is with tight coupling between language and audio signal processing framework. In version 3 McCartney moved SuperCollider to a loosely coupled design, separating the audio signal processing framework into its own distinct "SuperCollider Server" separate to the SuperCollider language SCLang.

The SuperCollider 3 server is a good example of decoupled interface design[57], and has been a popular framework choice for many live-coding environments. IXILang[53], Overtone, and Sonic PI[1], are just a few of the well known live-coding environments that provide language front-ends (i.e. are loosely coupled) to the Supercollider server.

The generally accepted wisdom in the software development community is that tight coupling is bad and loose coupling is good. However, while the added safety and security of inter-process communication is attractive, the significant downside to this approach is its limited flexibility "out of the box". An inter-process communication approach usually requires a considered, and generally task specific interface. This requires more work, on both sides of the interface. The fact that flexible coupling can generally occur *without any modification* of extant third party frameworks is a significant advantage for intra-process FFI solutions.

In practice both approaches are valuable and provide utility, and in Extempore's development it has often been valuable to begin with tight coupling and move to lose coupling overtime. In Extempore development it is often the case that tight coupling to a legacy C library proceeds a bespoke decoupled XTLang solution. The flexibility to move freely between these alternatives leads to a form of 'cannibalisitic' on-the-fly software development where new bespoke XTLang code is developed on-the-fly to replace C library code. Cannibalisitic in the loose sense that bespoke XTLang code is written on the fly to slowly cannibalise a tightly bound C library piece by piece.

3.1.3. Concurrency

Live-coding systems are fundamentally concurrent. On single-core systems, this concurrent behaviour must inevitably be realised by slicing CPU time into divisions small enough that many concurrent actions appear, to human perception, to be running in parallel. The trick is to ensure that each concurrent activity is given enough CPU slices, at the correct times, to maintain the fiction of continuity; *enough* precision, with the *correct* temporal accuracy.

There are two broad strategies for maintaining this fiction:

Pre-emptive multitasking where user-space programs are interrupted non-deterministically. The operating system will generally set an interrupt timer that forces control back to the scheduler at the end of each time slice. The behaviour is non-deterministic as the programmer has no guarantee about where or when his program will be interrupted.

Cooperative multitasking where user space programs are required to yield execution control back to the scheduler. The behaviour is deterministic as the programmer is responsible for detailing where in her program control is returned to the scheduler.

Since the later half of the 1990s pre-emptive multitasking has become the dominant strategy and is now the de-facto standard for all commodity operating systems. Pre-emptive multitasking systems are usually considered to be superior for general purpose computing because they remove the responsibility of program interleaving from the end-user programmer[97]. This removes a significant layer of complexity, making for a safer and fairer experience for all user-space programs.

For *general* operation, on consumer operating systems, the pre-emptive approach should provide a more reasonable level of service across a diverse range of applications; applications never intentionally designed to coexist. However, the cost of this fair scheduling is a limited ability to predict exactly when, and in what order, any particular piece of concurrent code will be executed. Additionally, the requirement to provide fair access to all is usually an impediment to specialised applications, such as real-time systems, which often depart significantly from the "general behaviour" expected of user space programs by the operating system.

Cooperative multitasking approaches are considered to be inferior for general purpose computing, because their concurrent processes must be *designed* to cooperate[97]. A cooperative general purpose operating system requires that all applications regularly yield control back to the operating system. The disadvantage being that a single nefarious application can lock up all system resources by failing to yield control at a reasonable rate. What is seen as a *reasonable rate* by one application domain, a word processor for example, may be completely unreasonable for another domain, such as an audio signal processing engine. There are good reasons why cooperative multitasking has fallen out of favour for general purpose scheduling. However, there is one very good reason why cooperative multitasking might be valuable for building cyber-physical systems - **timing**.

An advantage for cooperative multitasking systems, is that the application programmer

is given explicit, and deterministic (with caveats that will be returned to), control over the execution time of concurrent code. Making the application programmer responsible for choosing what code to run, at what time, and for how long, certainly places additional cognitive load on the programmer, but also provides the programmer with the ability to structure their code to better meet *timing* deadlines. Cooperative multitasking allows the programmer to structure code around timing, rather than throughput; to treat time as a correctness issue, rather than simply being a quality-of-service issue[50]. For cyber-physical systems, where latency is often more critical than bandwidth[4], the ability to manage the timing of code *as a correctness issue* is vital.

Supporting a cooperative multi-tasking approach in the design of a musical live-coding system is however complicated by the need to support commodity desktop operating systems. Windows, OSX and Linux are all pre-emptive multi-tasking operating systems. A hybrid approach is necessary in practice as most live-coding systems will ultimately run as a single, multi-threaded operating system process, i.e. with a single PID (process ID), and therefore be subject to commodity operating system scheduling.

Many-core systems provide an interesting opportunity for cooperative multitasking. As core numbers increase it is often the case that individual applications begin to completely consume processor cores - effectively a single end-user application per core (or multiple cores). This is becoming increasingly common as commodity operating systems begin to offer more advanced resource management, such as user space APIs for non-uniform memory access and processor core affinity. The advantage afforded by greater hardware parallelism is that the damage of a nefarious cooperative end-user applications will, to a larger degree, be limited to the complete consumption of only a single *assigned* core. Failure to yield in this context becomes largely a power/heat concern, rather than effecting overall quality-of-service, other shared system resources not withstanding.

3.1.4. Time

For a cyber-physical system to be relevant to the time-frame of its environment, it must have some means to measure time. "Real-time", as a computer science term, exists to draw a distinction between the time of an external environment, and the time of an internal computing system. The "real-time" is the measure of the computation's time in relation to some stream of external physical events.

In practice there are two primary ways to accomplish this. As Burns and Wellings[17, p.309] note:

- by having direct access to the environment's time frame;
- by using an internal hardware clock that gives an adequate approximation to the passage of time in the environment.

In the context of audio processing, the first approach is achieved by measuring the time frame of the incoming audio signal. Each sample, or more commonly each buffer of samples, can be used to drive a system interrupt. In the case of regular events, such as the buffers delivered in audio signal processing, these interrupts form a regular monotonic clock [4]. A clock with a direct relationship to the environment's time-frame. In this sense the cyber-physical system's "clock" is driven by the audio stream itself. Real-time becomes directly related to the physical phenomenon that is the core interest of the task domain – in this instance, sound.

The second approach relies on the granularity and regularity of an internal systems clock matching in some reasonable way to the external physical process of study. Although this "artificial" alignment is in some sense less satisfying than the first approach, it is the most common in practice. Indeed in the audio case the first approach is only

[4] Wall clocks, or calendar clocks, like the wrist watch, are not strictly monotonic (as they are subject to adjustments such as leap years etc.) but can be considered monotonic for our purposes here.

made possible because of the second approach. The *external* audio hardware's sensors and actuators, as well as the operating system's hardware abstraction layer, make the first approach available only because the second approach is actually managing all of the "sample clocking" behind the scenes.

While this might seem at first blush to make the first approach redundant, it is very often the case that the external hardware, which is often dedicated to high performance event processing, is considerably more effective at maintaining the high-precision and low-latency (i.e. granularity and regularity) clocking necessary for accurate measurement of real-world phenomena. Measurements that are, in the case of audio systems, based around the limits of hearing, which is at best around `20kHz`. As a point of reference, the highest standard sample resolution generally supported by modern audio devices is `192,000` samples per second. Although these samples are commonly buffered into groups of `64` samples, the interrupt rate required to meet real-time demand at this sample-rate is still `192,000/64=3000` interrupts per second; which is sub millisecond and well into the microsecond `kHz` range. Audio signal processing systems have quite stringent real-time requirements compared to many other cyber-physical systems.

In practice cyber-physical systems incorporate both approaches to clocking, and part of the responsibility of the cyber-physical system is to effectively manage the *many* distinct clocks that are likely to coexist in the system; regardless of which clocking strategies they utilize. Significantly complicating this are the inaccuracies inherent in all real-time (non logical) clocks. At any point in time there **is** inaccuracy between all of the real-time clocks in a given cyber-physical system.

Distinct from real-time clocks, and the internal cpu clock, are logical clocks. Logical clocks are not clocks in the traditional sense as they have no correlation to external physical events. Instead they represent logically consistent event orderings[46]. Logical clocks

are ubiquitous in computing systems where their unambiguous discrete representation makes them easy to reason about and to formalise.

The synchrony hypothesis[12], which assumes a logical clock driving a progression of inputs to outputs, is the foundation for a family of synchronous languages, both general purpose Lustre[68], Signal[49], Esterel[12], StreamIt[44], and from the computer music community ChucK[96]. In synchronous systems, events are processed through a directed graph in rounds. For audio systems, a single round would constitute the digital signal processing of an input sample (or buffer of samples), to an output sample (or buffer of samples). An essential assumption of the synchrony hypothesis is that the *work* done to transform inputs to outputs happens instantly (i.e. has zero computational cost). In this sense, the synchronous system is assumed to have infinite resources; infinite memory, infinite compute power, infinite I/O etc.. By making these assumptions, the synchrony hypothesis gives us simplicity, and a set of useful mathematical tools.

Synchronous systems line up very well with logical clocks. However, they do not always line up well with physical clocks; real-time clocks. One of the downsides to the synchrony hypothesis is that everything is expected to operate in lock step; everything happens within the monotonic round. There are various strategies for having graph elements skip rounds and alike but fundamentally synchronous systems are based around the idea of a single logical clock.

Where a round, a single run through the directed acyclic graph (DAG), has a relatively periodic execution schedule, and the worst case execution time (WCET) of the graph is of a shorter duration than the time between rounds, it is reasonably straightforward to match the logical clock of the synchronous system, to a physical real-time clock; and by association to the passage of real world events. If however the execution time of the DAG becomes increasingly non-uniform; perhaps different inputs have different

computational demands, perhaps the resources of the underlying hardware fluctuate, perhaps the garbage collector just kicked in; then the ability to match the logical time to the physical time becomes increasingly difficult. This situation is exacerbated as the computational load begins to approach some maximum threshold and the ability of the system to "catch-up" to the real-time clock becomes compromised.

An alternative strategy is to use a real-time clock to drive an asynchronous system of events. In a *timed multitasking*[52] approach events are time triggered tasks or actions. Trigger conditions for a task can be physical events, communication packets or messages from other tasks. What ties these events together is their assembly into a time ordered "scheduling" queue. Time ordered, as tasks are time-stamped in reference to a physical real-time clock. Messaging between tasks (where messages are themselves tasks) is asynchronously managed via the queue. All tasks are registered with an execution time (the task's real-time start time), and an execution deadline (the task's maximum acceptable run-time). For tasks that overrun their execution time, some form of exception handler is generally made available, allowing developers to manage task overruns on a per-application basis. This provides flexibility based on task criticality in varying domains.

For real-time systems, the advantage of the timed multitasking approach over the synchronous approach is that the internal timing of system events is directly tied to the natural progression of events in the physical world. Computational time is exposed, warts and all. Of particular relevance to the live programming context, timed multitasking is well suited to dynamic phenomenon and evolving aperiodic schedules[52].

In practice, it is often most effective to combine these two approaches into a hybrid design. It is clear that for some continuous, periodic systems, such as audio signal processing, for an uninterrupted supply of samples to be output at the correct times

(a necessity for uninterrupted audio), a requisite amount of computational work **must** be completed on time, and that this work is regular (monotonic). At least insofar as the system is capable of adequately meeting its functional requirements. For processing of this nature, the simplicity of the synchronous model is preferred, and in practice is almost universally adopted. Barkati et al., state that "In some loose sense, all music-specific programming languages use, in one way or another, synchronous idioms"[7, p.9].

However, there are computational demands, of a more ad hoc nature, that clearly do not fit well within the synchronous paradigm – loading large audio sample libraries from disk for example. Particularly where there is a substantial I/O component, or other significant computational overhead associated with the action. It is also the case that the synchronous paradigm is often encumbered when working with multiple clocks. Although multiple logical clocks can be partitioned it is often more practical to run multiple discrete synchronous DAGs, each with an independent logical clock (i.e. independent monotonic rate), and to manage these using a timed event model. A trivial example would be running a graphics DAG and an audio DAG in temporal isolation from one another with coordination managed by a timed event model.

A hybrid model supports *multiple* periodic, clock independent, synchronous processes operating within a system of asynchronous, potentially a-periodic, time triggered events. What enables these processes to cooperate is adequate clock synchronization and or translation and the introduction of *latency*.

3.1.5. Latency

Introducing latency into a system allows for a degree of clock uncertainty to exist between distinct processes, without unduly effecting the accuracy of timed output events. There is inherent instability between any real-time clocks running in a cyber-physical system.

Physical clocks drift[16], and logical clocks are only idealized clocks, that must be related to a physical clock at some point (i.e. audio frames to audio hardware).

The amount of acceptable latency, latency that does not adversely effect the outcome, is determined by context. For audio signal processing, acceptable latency is often quite low in comparison to other domains. In modern audio signal processing, a single sample (i.e. sample accuracy), is considered to be a necessity (or at least highly desirable). A significant advantage then for synchronous systems is that every input is synchronized with its output - they are atomic within a round. The introduction of asynchronous processes complicates event synchronization, as input is no longer temporally bound to output, and asynchronous tasks are not synchronized by a monotonic event cycle. To manage event synchronization in hybrid contexts it is necessary to introduce some degree of latency. By scheduling time-stamped events forward in time, it is possible for a real-time event driven asynchronous process to precisely (sample accurately in the audio context) schedule changes of state in a logically clocked synchronous process.

In practice, code cannot be written, parsed, compiled, and executed with zero temporal cost. Even factoring in an acceptable temporal cost (i.e. a temporal cost tuned to avoid discontinuities in human auditory sense perception), the time available is brief - in the very low milliseconds. By forward scheduling events it is as if the *present moment* has been pushed into the future, such that all events that should naturally have occurred in the present moment happen in the present moment plus some Δt. The magnitude of Δt is what enables the language run-time to parse, compile and execute code *before* the actions of that code are required. Due to the fact that all events are uniformly scheduled into the future a known latency is introduced, which is equal to *at least* Δt. A large Δt will ensure that there is time to manage the digital system's housekeeping before any scheduled events are required. However, a large delta will also increase the latency,

critically distancing the digital system from the flow of natural events in the physical environment. Cyber-physical systems attempt to balance these competing interests to provide the lowest possible latency.

Introducing a Δt latency will go some way towards managing the temporal cost associated with parsing, compiling and executing code, but has limited affect on the programmer's ability to "perform" temporal actions *in code*. Although one-shot calls can be used to "play" a system, there are obvious limitations regarding the effectiveness of this *direct* approach.

By adding the ability to explicitly schedule events into the future, the programmer is able to compose future actions into computational processes that can be pro-active in their realisation. By being pro-active in this manner the programmer is free to concentrate on other necessary tasks - such as composing other pro-active computational processes.

3.2. Full-stack live coding

The proceeding section highlighted some of the major design considerations when implementing a half-stack live programming programming environment. This section explores the design of a full-stack live programming environment – a cyber-physical programming environment.

Where the half-stack design focused on the high-level *live orchestration* of an extant real-time systems framework, the full-stack design focuses on the low-level *live construction* of a real-time cyber-physical system. Where the half-stack live programming environment provides an external interface to an extant real-time system, the full-stack live programming environment is capable of building the real-time system from the ground up. In this sense, full-stack live programming is inherently building from within. Full-

stack live programming lifts itself up from its own bootstraps.

Full-stack cyber-physical programming is based on the premise that it is valuable, to not only *control* real-time frameworks at run-time, but also to *design, build and modify* real-time systems at run-time. To build systems designed to operate in the environment from within that self same environment. From the design perspective this introduces the challenge of designing a programming language that is easy to reason about (cyber-physical programming is a demanding cognitive activity) whilst at the same time being *as efficient* and *as flexible* as a systems language.

3.2.1. Leaky Abstractions

The compromise promoted by the half-stack design is that high-level programming languages can be used for day-to-day productivity, leaving the framework programming, in low-level languages, to "experts". While this strategy has enjoyed great success – in scientific computing R, Python (NumPy, SciPy), Matlab and Mathematica provide strong examples – there are clear limitations to this approach.

By way of illustration, the R language is a high-level language whose wider environment, the R environment, has been designed from the outset to interface with native code libraries written in C. R is a classic half-stack live programming environment coupling the high-level R language, with a substantial numerical stack of library code written in the C/Fortran languages. R provides the high-level orchestration environment, while C/Fortran provide the low-level computational performance (the extant framework). R requires this native numerical stack in order to provide adequate numerical performance, as its own interpreter is too slow for heavy numerical work[61].

The idea behind the high/low compromise is that R programs will call across the C/Fortran FFI whenever heavy numerical lifting is required. In principle, the relative

inefficiency of the R language should be ameliorated by the high efficiency of the systems code (C/Fortran) layer. The expectation being that the majority of CPU time will be consumed in native code, where the computational heavy lifting can be efficiently managed.

However, contrary to expectations, when analysing a corpus of some 3.5 million lines of R code Morandat et al. noted the relatively low amount of time spent inside the C/Fortran FFI.

> Given the nature of R, many numerical functions are written in C or Fortran; one could thus expect execution time to be dominated by native libraries. The time spent in calls to foreign functions, on average 22%, shows that this is clearly not the case.[61, p.17]

Separating high-level orchestration from low-level numerical performance is inherently leaky. Algorithms requiring high-performance inevitably bubble-up into the scripting layer. Live programming makes this situation even more problematic by encouraging deeper algorithmic entanglements between the scripting layer and the native layer. Where should the script stop and the heavy lifting start? Indeed cyber-physical programming, by its very nature, strongly encourages a bubble-up programming approach. Particular attention is paid to this bubble-up live programming style in the particle in cell physics case study in Chapter 5.

Full-stack cyber-physical programming endeavours to provide the same high performance *at the scripting layer* as is expected at the systems layer. Indeed full-stack cyber-physical programming is predicated on the idea that there should be no distinction between programming at the scripting layer and programming at the systems layer.

3.2.2. The Language Interface

The live programmer's interface is code - a textual language used to describe a computational form. The text of this language, provides an abstract description of a computational process that must, at some *future* time, be executed on some physical hardware. One important function of the source code's *text* is to inform the programmer about how the program will behave at run-time. This information is used by the programmer to construct a mental model - to reason about the code in some meaningful way.

There are many possible ways in which a programmer may reason about a given program's text. Alan Perlis famously quipped that "Lisp programmers know the value of everything and the cost of nothing" (Epigrams on Programming[66]). Whilst all programmers develop mental models for the run-time execution of their code, their mental model is skewed by the language designer. When Haskell programmers declare that Haskell is a superior language for *reasoning* about code, they are specifically talking about *equational reasoning*. The designers of Haskell made specific choices that were intended to make it easier for programmers to build a mental model that correlates more soundly to an equational/mathematical model.

As Perlis observed, skewing the model in one direction, often has consequences in another. For Haskell, design decisions that made the language easy to reason about from an equational reasoning perspective (e.g. immutability and laziness), also make the language more difficult to reason about from a time and space perspective[5].

Systems languages such as C make an alternative trade-off, providing the programmer with a mental model more aligned to the machines physical hardware architecture than the more abstract models commonly favoured by many functional and dynamic languages (including the Lisp dialects alluded to in Perlis's quip). The C approach en-

[5] For a pertinent discussion of the performances costs associated with immutability and laziness in Haskell see [18]

ables programmers to more easily reason about the time and space costs associated with their programs, by providing a programming interface (i.e. the language) that presents a model that more closely relates to the physical architecture of the machine.

A ramification of making the C language easier to reason about from a hardware perspective is that the C language is usually described as being less easy to reason about from a task domain perspective. While C makes it easier to write code that directly targets the hardware the imperative style that this approach entails is necessarily more verbose than Haskell's more declarative style.

For a full-stack cyber-physical programming environment, the design trade-offs are particularly challenging. Cyber-physical programming is a practice undertaken in a rapid development environment where a programmer's ability to reason about their code is often time sensitive - very often at the boundaries of human perceptual abilities. A consequence of this time sensitivity is that there is often very little time to build reasonable mental models about source code in any particular dimension. The Haskell programmer may be willing to accept that it will take her a little longer to reason about her programs execution performance profile, because that time is available to her. A C programmer may accept that it takes him longer to correctly model a given task domain, but once completed that program will execute efficiently. Unfortunately for the live-coder all dimensions of source code reasoning are equally time dependent.

A key challenge then, in the development of a cyber-physical programming environment, is to provide a language interface (i.e. a programming language) that can be easily reasoned about, within the temporal bounds common to many physical tasks, across a number of dimensions of reasoning. A language interface for cyber-physical programming must try to balance the "equational reasoning" of a Haskell with the time and space "cost reasoning" of a C.

3.2.3. Designing for *reasonable* performance

As physical computing machinery becomes increasingly complex, and the physical limitations of computation increasingly constraining, the ability for a programmer to wring every last cycle from her hardware becomes more challenging.

One approach is to keep throwing more hardware at the problem. The general idea being that it is cheaper (both socially and economically) to "waste" CPU cycles using higher level, but relatively inefficient programming environments, rather than "wasting" valuable human cycles working with lower level, although more efficient programming environments. The intimation being that an inefficient programming environment operating at a higher level of abstraction should display less cognitive dissonance (i.e. should be better focused on the domain problem) than a lower level, but presumably more efficient, environment.

Unfortunately, where hardware is already size and heat/power constrained throwing hardware at the problem often makes the problem increasingly intractable. It is watts, rather than pure numerical performance per se, that is now driving both the development of the next generation of exascale Supercomputers[38] at one extreme, and IoT[31] (the Internet of Things) at the other. It seems clear that simply throwing more hardware at the problem, when the problem itself has hard physical limitations, like heat/power and size, simply exacerbates the problem.

If low-level system performance and efficiency goals cannot be adequately attained – by throwing hardware at the problem – then perhaps it might be possible instead to rely on a sufficiently smart compiler. A sufficiently smart compiler capable of abstracting away low-level details without sacrificing low-level flexibility or performance. The form of a "sufficiently smart compiler"[33] changes over time, but at this moment in history large virtual machines like the JVM and the CLR are the usual manifestation.

Advocates of sufficiently smart compilers highlight that declarative solutions require less work by the programmer and ultimately perform better, as the compiler is better suited to managing the low-level imperative bookkeeping – state management, register allocation, parallelisation etc.. Because declarative programming *describes* rather than *instructs*, it is argued that the compiler is better able to perform advanced optimizations. A garbage collector can better manage memory, a profiling JIT compiler can better optimize native code and the compiler can parallelise more effectively using vector registers/opcodes.

It is difficult to argue against the sufficiently smart compiler; after all, no one wants to be distracted by irrelevant imperative bookkeeping! However, from a *system's programming perspective*, the sufficiently smart compiler breaks-down in a couple of key areas. "Smart language" technologies, including garbage collection, and JIT compilation, optimize for, and are themselves optimized for, the general case. If you fit the general case, well and good. If you don't fit the general case, then you are at best at a disadvantage, and at worst flat out of luck. In "Scalability! But at what COST?"[59], McSherry et al., discuss some of the hidden costs associated with "smart" distributed parallelisation, with a particular focus on graph processing.

> Modern JIT compilers, such as those in virtual machines like Oracle's HotSpot for Java or Google's V8 for JavaScript, rely on dynamic profiling as their key mechanism to guide optimizations. While these JIT compilers offer good average performance, their behaviour is a black box and the achieved performance is highly unpredictable.[74, p.41]

For the systems programmer, the inherent unpredictability of "smart" technologies is often *unreasonable*, in both senses of the word. It is not the *average performance*, but the *worst case performance*, that is usually of primary concern in systems programming

contexts. Additionally, the black-box nature of smart compilers makes reasoning about a source codes run-time performance challenging, often extremely so. A well known aphorism is that "all problems in computer science can be solved by another level of indirection" to which is added the often cited corollary "except of course for the problem of too many indirections"[6].

The two most common approaches to circumventing these *unreasonable* performance issues when adopting high-level languages, for low-level systems programming, is either to adopt an FFI style model (as discussed in the half-stack section) or alternatively to extend the high-level language down to meet low-level requirements.

There are many good reasons for wishing to move systems programming up the abstraction tree. As noted by Frampton et al. "The power of high-level languages lies in their abstraction over hardware and software complexity. Leading to greater security, better reliability, and lower development costs"[27, p.81]. However, the abstractions required to achieve these goals are often antithetical to the goals of systems programming. Frampton et al. go on to say "However, opaque abstractions are often show-stoppers, for systems programmers, forcing them to either break the abstractions, or more often, simply give up and use a different language"[27, p.81]. Many of the features that define higher-level languages - boxed data types, garbage collection, read/write barriers, array bounds checking - are exactly the features that are most intrusive for low-level programming. The trouble is that for the systems programmers, what is considered *irrelevant bookkeeping* is very often at the very heart of the matter!

Some impressive attempts have been made to address these concerns, for example the Jikes RVM (research virtual machine) **org.vmmagic** package, which incorporates compiler intrinsics and semantic regimes, provides one such low-level system's program-

[6] Although commonly attributed to either David Wheeler or Andrew Koenig, there appears to be no definitive attribution for this aphorism, or its corollary, which is sometimes attributed to Rob Pike.

ming solution for Java[27]. However, these solutions generally provide only partial answers (org.vmmagic supports unboxed primitive types for example, but not unboxed compound types), and at the cost of significant complexity. Complexity for both the language designer/implementer, but also for the end-user programmer, who is faced with a considerable stack of abstraction layers to be managed (see org.vmmagic semantic regimes). There is also a question mark over the true extensibility of these platforms. The org.vmmagic project strives to provide an extensible framework for low-level systems programming, but relies heavily on an extensive library of built-in compiler intrinsics. These intrinsics must be maintained and extended by the Jikes RVM core compiler team, making more work for both the system's developers, who must define new intrinsics to support new hardware features, and end-user programmers who must strive to understand the semantics of these "new" intrinsics and their relationship to the underlying hardware that the "new" intrinsics support.

Ultimately smart compilers are necessarily designed for the general case, where systems programming is often about unique and specific cases. Smart compilers are relatively inflexible, where systems programming requires great flexibility. Smart compilers are difficult to reason about, from a low-level systems perspective. And finally, smart compilers are complex to build, configure and operate, making them difficult to optimally configure, and to port to new hardware platforms.

3.2.4. "Live" cyber-physical programming systems

The task domains that have been explicitly identified to this point (simulation code, audio-signal processing, computer games, graphics processing, real-time systems), share the common notion of a central monotonic cycle. A game engine will continue to update the scene, despite the fact that the player has not moved; fire embers rise, shadows dance

as the fire flickers, an AI character moves "into shot"; a digital synthesizer must work to generate a continuous signal, just as a voltage controlled analogue synthesizer would be expected to do; and a computational simulation will continue to integrate indefinitely.

These systems are expected to complete a certain amount of work per cycle, and that work is *reasonably* consistent. A certain number of polygons must be drawn, a certain number of oscillators must be summed, and a certain number of particles must be integrated. The search for increased fidelity is never complete; we always want more polygons, more oscillators and more particles. Fidelity ensures that computer games programmers, audio DSP programmers and HPC programmers are always looking for ways to extract every last cycle of performance from their current hardware – which is also usually at the forefront of what is possible.

A distinguishing feature of this monotonic style is a heavy reliance on straight ahead numerical performance. Whilst games developers, audio developers and HPC developers are deeply interested in efficient algorithms, multi-core applications, and distributed processing, what distinguishes this group is their low-level interest in memory locality (cache coherence, scratchpad utilisation, NUMA) and CPU optimisation (especially SIMD).

What distinguishes cyber-physical programming from more general systems programming is the programming experience. Cyber-physical programming extends systems programming to support *live edit*, and to generally support a more dynamic, exploratory and ephemeral, programming experience. Where systems programming is strongly oriented towards the engineering of a fixed product, cyber-physical programming is fundamentally more experimental in nature. Cyber-physical programming *does not preclude production* but supports a more experimental approach to its realisation. A cyber-physical programmer may explore the development of a serial protocol by building that protocol on-the-fly, testing for correctness along the way by watching the system's *ongoing* real-time output

via an oscilloscope. Any code written while developing (exploring!) the protocol *may* be kept for inclusion in some future production, but may also be used ephemerally, for purely design/research purposes.

The necessity to support *live edit* brings the tooling of systems programming into the domain of real-time systems. Compilers, linkers, debuggers etc., are subject to the temporal bounds of the task domain. An audio programmer may wish to compile in a filter change to match the song's chorus. Compiling and hotswapping in this source code change, with respect to the temporal demands of the musical performance, is a novel concern of cyber-physical programming systems. This is a *functional requirement* not a quality of service issue.

3.2.5. Code specialisation

Straight ahead numerical performance is an area where higher-level languages are often at their greatest disadvantage. A common practice in higher-level languages is to use a uniform data representation known as a boxed type. A boxed type is a reference data representation that wraps some other data-type (a primitive type, or some other boxed type), incurring both a size (requires more space) and performance (an extra level of indirection) overhead.

The uniform size (i.e. typically a pointer) of boxed types provides many advantages when implementing language features such as parametric polymorphism and garbage collection. However, where high performance is required the cost of storing (contiguous objects in cache) and un-boxing (additional redirections) primitives is significant. For this reason, languages such as C#, Java, JavaScript and Haskell, go to great lengths to optimise away boxed types at compile time.

Generally speaking modern compilers do a good job of automatically un-boxing prim-

itive types, particularly in the context of numerically heavy "hot-loops". JIT compilers are known to provide well optimised native code[32]. The standard JIT model injects profiling code into an initial un-optimised compilation pass; which may directly output *un-optimised* native-code (e.g. heap allocated local variables). This un-optimised code is then run for several passes as profiling information is captured. This profiling information is then used to identify sections of code that would benefit from further optimisation. These "hot" sections are then optimised, and replaced on-stack at some safe point. This process of optimised refinement often proceeds through multiple levels of optimisation as increasingly sophisticated profiling information is attained.

Unfortunately, for real-time systems, the time required to 'spin up' optimised code is often unacceptably long (and lumpy!), with less than fully optimal code running during this initial profiling phase[74]. Compounding this problem, live programming encourages programmers to constantly modify the behaviour of their code - in turn forcing the JIT to de-optimise (and then to re-optimise) code regularly. Not only is this costly from a pure efficiency standpoint, but it also results in temporal indeterminacy that will be unacceptable in many real-time contexts. For example, for an audio signal processing system, the time required to reach optimised code will very likely result in glitching in the audio stream for any changes made to DSP related code. In a live programming context this is very likely unacceptable as these phases occur regularly, as the live programmer continues to make on-the-fly modifications to extant DSP routines.

In practice, many tracing JIT environments choose to "warm up" their systems, by treating the system to a range of prepared input data in preparation for "going live". Although not practical for live programming (as live programming is constantly adding new code), an alternative strategy is to run the byte-code path in parallel while the JIT optimises new code paths, only switching the primary code once an optimised code path

is available. However, this inevitably results in increased latency and additional power and performance overheads, as well as additional complexity.

Ultimately efficient JIT compilers are difficult to implement, and complex to reason about, from an end-user performance perspective. More specifically, JIT compilers make it more difficult for the programmer to reason about native code generation. When and how will the JIT optimise *this particular* code path and what level of optimisation is in effect, for a specific code path, at any given time?

3.2.6. Regional Memory Management

Due to the high performance profile, low-tolerance to temporal jitter, and relatively monotonic nature of many real-time cyber-physical systems, garbage collection remains relatively uncommon in practice. As Boyapati et al. state in relation to the Real-Time Specification for Java (RTSJ) "real-time threads cannot use the garbage-collected heap because they cannot afford to be interrupted for unbounded amounts of time by the garbage collector"[13, p.324].

Instead it is common for these types of systems to employ a custom memory allocator, very often employing some variation of regional memory management[90] (also known as memory zones, havens, pools, arenas).

During the 1990s Tofte and Taplin explored an extended notion of the stack discipline based around a stack of regions. Region based memory is allocated and de-allocated in a stack-like manner, although the lexical extent of a region is more flexible than a traditional function call stack frame. Tofte and Taplin's work on regional memory management in ML was designed to be fully automatic and verified statically. In particular Tofte and Taplin were concerned with regional type inference for memory verification[89].

Some of the benefits provided by a regional approach are; trivial pre-allocation of

partitioned memory; extremely fast allocation and reuse (write-over); flexible and fine-grained control; and location awareness[40, 62, 29, 13]. Taken as a whole, the significant advantage is that allocation and de-allocation can be performed in predictable time.

Tofte and Taplin's work on regional memory management has been influential; particularly their focus on automatic inferred type analysis for static verification. However, in "A Retrospective on Region-Based Memory Management" Tofte et al. state that:

> What has emerged is that the very heavy employment of automatic program analyses has pragmatic drawbacks and also that the implementation of regions, once it gets all the way down to the machine representation, becomes somewhat clumsy[91, p.261].

Tofte et al. suggest that a possible strand of future work, "is to use the accumulated knowledge about regions in the design of a new programming language that allows programmers to program explicitly with regions"[91, p.261]. This is the approach that has been adopted in the development of Extempore.

The implementation of a live language 4

Extempore is a novel new live programming environment designed for cyber-physical programming. At Extempore's heart is the new XTLang (a contraction of Extempore Language). Originally designed as a domain specific language[79] for audio signal processing in the author's Impromptu live-coding environment[78], XTLang has subsequently developed into an independent, general purpose, full-stack cyber-physical programming language.

The development of Extempore, and XTLang more specifically, has been pragmatically motivated by a desire to support live programming in real-time systems contexts. Idiomatic XTLang code, should be easy to reason about, from a performance perspective, but should still provide the higher level language features expected of a "modern" programming language. It is not enough to simply *be* a live programming environment, Extempore must also prove to be an effective general purpose programming environment, supporting the "expressive programming"[94] language features and tooling expected in modern development practice.

This chapter provides an overarching description of Extempore's design and implementation, with a particular focus on the new Extempore language, XTLang.

4.1. Architectural Overview

Extempore is, in essence, a project made up of three primary parts.

A Scheme language interpreter written in C, conforming to the Scheme R5RS specification. Extempore's Scheme interpreter was seeded from Impromptu[78], which

was in turn seeded from TinyScheme[82]. The Extempore Scheme interpreter supports a C FFI and a top-level REPL.

An XTLang compiler which is made up of a front-end and a back-end. The front-end of the compiler, which is written in Scheme, translates XTLang s-expressions into Low Level Virtual Machine[1] (LLVM) IR strings. The compiler back-end, which is written in C++, compiles LLVM IR strings into native code. The front-end of the compiler is developed as a substantial part of the Extempore project, and the back-end of the compiler is developed and distributed as a part of the LLVM project[47]. The LLVM backend(s) support native code compilation in-memory, or to disk, and supports a growing number of architectures including, ARM, x86, x86-64, PowerPC and PTX.

A network addressable run-time service supporting process management, LLVM backends and network communications. The Extempore run-time infrastructure is broadly responsible for the scheduling and interpretation (including compilation) of incoming s-expressions (either Scheme or XTLang), on one or more Extempore processes, running either locally or remotely.

While the **Extempore run-time** runs as a single operating system process (i.e. Unix process), an **Extempore process** *is* a Scheme interpreter. Individual Extempore processes maintain their own unique Scheme language memory address space.

Each Extempore process is network addressable via an asynchronous messaging thread. This allows programmers (their text editors that is) to send arbitrary Scheme s-expressions to one, or more, Extempore processes (i.e. Scheme REPL) running locally

[1] Despite the name, LLVM is not really a virtual machine but is instead a library for designing and building compilers. Developed by Chris Lattner, and others, at the University of Illinois in the early 2000s, LLVM has gone on to become one of the most substantial compiler infrastructures available, with major investment from computing companies such as Apple, IBM and Nvidia[5].

or remotely. Additionally, Extempore processes use these channels for inter-process message passing. An Extempore process makes no distinction between local and remote communication.

The Extempore run-time is able to spawn any number of Extempore processes, but by default initializes two — a *primary process* and a *compilation process*. The primary process is designed for general purpose computing; the evaluation of arbitrary Scheme language s-expressions. Ultimately there is nothing "primary" about the primary process, and an Extempore session may end up running many such processes; each with a unique name, second, third, Fred, Wilma etc.. Unlike the primary process, which is a general purpose process, the compilation process is dedicated to a single task, the compilation of XTLang expressions.

The compilation process, in all other regards a standard Extempore process, is a Scheme REPL that accepts strings of XTLang and translates that code into LLVM IR strings. This translation is carried out by the XTLang compiler program, which is itself written in Scheme. The compilation process is responsible for running the XTLang compiler program, which accepts XTLang expressions as input and produces LLVM IR as output. Once a valid LLVM IR string is produced, the XTLang compiler sends that string onto the appropriate LLVM back-end via Extempore's Scheme C FFI. The native LLVM back-ends, which are statically linked into the Extempore run-time system, are responsible for compiling LLVM IR into executable native code blocks, which are written in-memory. These in-memory executable code blocks are available to any XTLang call site (i.e. to other compiled XTLang code), as well as to Scheme calls via the Scheme C FFI. In-memory native code compilation with small compilation units is often seen as something of a mystery, and yet the process is straight forward.

The XTLang compiler (the Scheme program) is itself broken into two parts, a front-end

and a back-end, with its own Intermediate Representation (XTLang IR). The XTLang compiler front-end, the semantic analysis section of the compiler, is responsible for globalization (i.e. for free variables), lambda lifting, scope and environment analysis, closure conversion, assignment conversion (i.e. to SSA form), type inference and static type checking. The front-end of the compiler outputs XTLang IR in static single assignment (SSA) form. The back-end of the compiler accepts this XTLang IR and performs a relatively straightforward transformation into LLVM IR [2].

Listing 1 through 4 show the progression of a simple XTLang function from XTLang code, into XTLang IR, then into LLVM IR, and finally into x86 assembly. The output presented is taken unmodified from the XTLang compiler, with the exception of some rudimentary formatting, and the addition of some minor ;; comments.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; XTLang Function

(bind-func times2 (lambda (x) (* x 2)))
```

Listing 1: XTLang source code

[2] It would be relatively straightforward to write a bespoke X86 or ARM assembly language backend for the XTLang compiler. Thereby removing Extempore's dependence on the LLVM project. A simple (i.e. non-optimizing) x86 back-end would be fairly straightforward to implement, as most of the complexity is in the XTLang front-end. However, writing an optimizing x86 compiler back-end, such as that provided by LLVM, would be a much more substantial project.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; XTLang IR (statically typed polymorphic closure)

(__make-env 3
            (((times2_adhoc_W2k2NCxpNjRd 213 2 2)
              (__make-closure-z 0
                                "times2_adhoc_W2k2NCxpNjRd__264"
                                2
                                ((times2_adhoc_W2k2NCxpNjRd 213 2 2))
                                ((x . 2))
                                (begin
                                  (ret-> times2_adhoc_W2k2NCxpNjRd (* x 2))))))
            (begin
              (ret-> times2_adhoc_W2k2NCxpNjRd
              times2_adhoc_W2k2NCxpNjRd)))
```

Listing 2: XTLang Intermediate representation

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; LLVM IR (C-ABI function with memory zone and environment)

define dllexport fastcc i64
       @times2_adhoc_W2k2NCxpNjRd__264(i8* %_impz, i8* %_impenv, i64 %x) #1
{
entry:
  %val267 = shl i64 %x, 1
  ret i64 %val267
}
```

Listing 3: LLVM Intermediate Representation

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; x86 ASM output (memzone and environment are optimized away)

#
# times2_adhoc_W2k2NCxpNjRd__264
#
0x11dc5b010: 8c212048d48  leaq  <mem:(<reg:%rdx>,<reg:%rdx>)>, <reg:%rax>
0x11dc5b014: 8c2  retq  <imm:\$0x8>
```

Listing 4: x86 Assembly

The reader may be questioning why the XTLang IR is so verbose given the relatively succinct LLVM IR output. While Listing 1 does present an accurate picture of the compiler's progression, it only presents a partial picture. In fact the compiler builds, not only one target, but several. The target presented here `times2_adhoc_W2k2NCxpNjRd__264`, is the function worker, the **worker target** in XTLang parlance, the primary call destination of an XTLang call site, such as `(times2 3)`. However, XTLang functions are closures, and extra machinery is required.

In addition to generating a worker for `times2` the XTLang compiler also generates a **maker target**, for constructing a `times2` closure, a **native target** which enables C functions to call XTLang **worker targets** directly (i.e. allows C to call XTLang *closures* directly), and a **scheme target** which automatically manages Scheme-C-FFI conversion.

In total, a single XTLang function definition like `(bind-func times2 (lambda (x) (* x 2)))` results in the generation of these four native C-ABI functions - the **worker target**, the **maker target**, the **native target**, and the **scheme target** - where the **native target** and the **scheme target** are responsible for calling into the **worker target** from C and Scheme respectively, and the **maker target** is responsible for the construction of closure environments (the environment) which include a reference to the

worker target (the function). All functions in XTLang are closures (a function and its environment).

4.2. Process Model

Extempore processes are heavyweight, requiring three core operating system threads per Extempore process.

1. A thread dedicated to executing top-level s-expressions in an Extempore Scheme language interpreter. Each Extempore process has a dedicated Scheme interpreter, with its own scheme object heap memory, stack register and instruction pointer.

2. A dedicated thread running a parallel garbage collector for the exclusive use of the Extempore processes Scheme interpreter. Extempore's Scheme language garbage collector is a custom, concurrent and incremental, variation of Henry Baker's four colour treadmill[6][77].

3. A network server thread to manage all network connections to a given Extempore process. Each Extempore process can be identified by its IP address and TCP port number and is responsible for marshalling all events (i.e. s-expressions) sent to the process.

Events sent to an Extempore process arrive as plain text s-expressions [3]. Each top-level s-expression is given a timestamp on arrival and persists in a dedicated Extempore process queue. This Extempore process queue is interrogated by the scheme interpreter thread, with s-expressions popped from the queue and evaluated as soon as possible. S-expressions arriving at the Extempore process are not directly scheduled (i.e. the queue

[3] Extempore provides no security for these connections, which given Extempore's cyber-physical nature is obviously of some concern. It is common practice to run these communications over secure tunnels, although out of the box security is planned for future release.

is not priority ordered), but are instead executed as soon as possible in arrival order. Any number of network connections (either local or remote) are aggregated into a single work queue, and processed serially, in arrival order.

As well as queuing s-expressions, the process server work queue is also able to store Scheme Objects such as Scheme closures, Scheme continuations, and Scheme C FFI calls. Scheme objects are added to a work queue from the work queue's own interpreter. Any code executing in the Extempore interpreter is capable of adding additional items to the Extempore process work queue. These work queue items are interleaved with incoming network events, with all events processed in strict serial order.

The Extempore process work queue forms the foundation of an event driven reactive programming[12] model, with new events (expressions, closures and continuations) processed in serial order. However, as Extempore processes are non pre-emptive, and events are executed *to completion* in serial order, the ability for the system to adequately react to new events, as they arrive, is predicated on the process's ability to execute events in a timely manner.

Where an event in an Extempore process is ultimately an arbitrary piece of code (i.e. not known ahead of time), it is difficult to conceive of a theoretical model that could adequately prove the temporal consistency of an Extempore process. Instead, Extempore relies on the programmer to reason about the performance cost of the events in the system at any given point in time, and to construct their events appropriately. Extempore supports the programmer in two ways. Firstly, Extempore allows programmers to attach a timeout to their events (to their code execution), such that late running events terminate early, via an exception handler. Although terminated events will degrade the performance of the system, early termination will ensure that reactivity is maintained. For example, an infinite loop that would otherwise block the process indefinitely will

terminate upon breaching its timeout. Secondly, and more fundamentally, as a cyber-physical programming environment, in which the programmer is an *active* participant, the programmer should notice degradation in the performance of the system, and be able to adjust the system to adequately address the shortcomings. Of course the more information the system can provide to the programmer, graphical feedback of the state of the work queue, conservative timing analysis of individual work items etc., the better (Some of these ideas are discussed in previous publications[77, 86]). However, ultimately the programmer takes responsibility for ensuring that a given processes work events are manageable.

The reactivity of a cyber-physical system is domain dependent. An acceptable rate of event processing (or more problematically event blocking) in one domain may be completely unacceptable in another. Moreover, there are likely to be multiple rates of reactivity running concurrently in any given domain, naturally or artificially imposed. Extempore is able to support concurrent rates of reactivity by spawning additional general purpose Extempore processes, where each Extempore process maintains its own event queue and parallel execution. The programmer is then able to assign long running events to "slow reaction" processes and shorter time sensitive events to "high reaction" processes. A common Extempore programming idiom is to populate a secondary "offline" process, for long running computations. The "offline" process can then message the "high reaction" process with event results. Extempore's Interprocess Communication Processing (IPC) framework makes this communication straightforward.

```
;; spawn two new Extempore processes
(define proc1 (ipc:new "online" 7089))
(define proc2 (ipc:new "offline" 7088))

;; reads a sample every second
;; accumulates five frames of data for 'offline' processing
(define fast-work
  (lambda (frame)
    (print (ipc:get-process-name) ">> read sensor frame" frame "\n")
    ;; every 5th sample process some data in 'slow-work' on proc2
    (if (= 0 (modulo frame 5))
        (ipc:call-async proc2 'slow-work frame (range frame (+ frame 5))))
    (sys:sleep *second*)
    ;; and repeat
    (fast-work (+ frame 1))))

;; print processing results
(define print-results
  (lambda (frame result)
    (print (ipc:get-process-name) ">> summed frame group" frame "=" result "\n")))

;; 'slow-work' on proc2 does not interfere with 'fast-work' on proc1
(define slow-work
  (lambda (frame data)
    (print (ipc:get-process-name) ">> processing frame group" frame "\n")
    (sys:sleep (* (random) 10.0 *second*))
    ;; print results on proc1
    (ipc:call-async proc1 'print-results frame (apply + data))
    'done))

;; start 'infinitely recursive' fast-work on proc1
(ipc:call-async proc1 'fast-work 0)
```

Listing 5: Reactive online/offline event processing

Listing 5 leaves an open question around the processing of `print-result` events. As it stands, `print-result` events must be interleaved into the `proc1` work queue; interleaved with recursively generated `fast-work` events. What makes this possible is `sys:sleep`. `sys:sleep` yields the execution of fast-work (the `fast-work` event) and simultaneously places a new anonymous continuation event onto the current Extempore processes work queue. When resumed `fast-work` will continue to execute recursively as defined. In this way pseudo `fast-work` events are freely interleaved with `print-results` events, and `proc1` reactivity is maintained.

In practice `sys:sleep` does not directly place the anonymous continuation event onto the work queue, but instead timestamps the continuation event with the requested sleep time, and submits the event to a priority scheduling queue. This real-time Earliest Deadline First[17] (EDF) scheduler is ultimately responsible for pushing events onto Extempore process work queues at the exact time specified by the events timestamp. In this sense Extempore supports both; **immediate events**, which are events pushed directly onto an Extempore process work queue and; **timed events**, time-stamped events pushed to an EDF priority scheduler, and at the appropriate future time, pushed onto an Extempore processes work queue.

While Extempore supports a number of ways to deploy an event, the most common are `ipc:call` and `ipc:call-async` (for immediate events) and `schedule` or `callback` for timed events (where `callback` is an alias for `schedule`). A more idiomatic implementation of `fast-work` would use `schedule` rather than `sys:sleep`. This is the form of a **temporal recursion**[78], Extempore's idiomatic cooperative concurrency paradigm.

4.3. Temporal Recursion

Temporal recursions share much in common with co-routines, and have in various guises been used since the 1960s. Within the audio and music communities the most well known precedents are Collinge's Moxie[19], Dannenberg's CMU MIDI toolkit[21], and McCartney's SuperCollider[57][4]. Impromptu[78] and subsequently Extempore, have extended these ideas incorporating a richer application of temporal recursion; most significantly the application of continuations to temporally recursive structures[77].

A temporal recursion is most simply defined as any block of code (function, method, etc..) that schedules itself to be called back at some precise future time (where precision is related to latency). A standard recursive function could be considered to be a temporally recursive function that calls itself back immediately, with zero temporal delay.

[4] To the best of our knowledge the first use of the term "temporal recursion" was by the author in [78]

```
;; A standard recursive function
(define my-func
  (lambda (i)
    (println 'i: i)
    (if (< i 5)
        (my-func (+ i 1)))))

;; A temporally recursive function with 0 delay
;; (callback (now) my-func (+ i 1)) ~= (my-func (+ i 1))
;; (now) means immediately
(define my-func
  (lambda (i)
    (println 'i: i)
    (if (< i 5)
        (callback (now) my-func (+ i 1)))))
```

Listing 6: Recursions

In Listing 6 `(callback (now) my-func (+ i 1))` serves a similar function to `(my-func (+ i 1))`; both are responsible for calling back into `my-func` *immediately*, passing an incremented value for `i`. However, the way in which these two recursive calls operate is substantially different. The temporal recursion, that is formed by the recursive call `(callback (now) my-func (+ i 1))`, is implemented as an event *distinct* from the current execution context. While the call `(my-func (+ i 1))` maintains the control flow, and potentially (assuming no tail optimisation) the call stack, the `(callback (now) my-func (+ i 1))` schedules `my-func` and then breaks to the top level, at which point control flow is handed back to the real-time scheduler.

One pragmatic distinction between a standard recursion and an immediate (no temporal delay) temporal recursion is that the temporal recursion does not keep adding new

stack frames to the call stack. Because of this, a temporal recursion must maintain any argument values that are passed (maintained) by the temporal recursion. The arguments must be maintained by the temporal recursion *event*.

The distinction between standard and temporal recursions becomes more obvious once the temporal recursion is scheduled beyond `(now)`. Because `callback` returns control back to the scheduling engine, it is possible to interleave the execution of multiple temporal recursions *concurrently*. Listing 7 interleaves two temporal recursions. First `a` runs then `b` then `a` then `b` etc..

```
(define a
  (lambda (time)
    (println "I am running a")
    (callback (+ time 40000) a
              (+ time 40000)))) ;; argument time

(define b
  (lambda (time)
    (println "I am running b")
    (callback (+ time 40000) b
              (+ time 40000)))) ;; argument time

(let ((time (now)))
  (a time)
  (b (+ time 20000)))
```

Listing 7: Temporal Recursion

In Listing 7 `time` is passed as a parameter to both function `a` and function `b`, with both `a` and `b` incrementing time at the same rate. The interleaving is defined by the initial offset of `20000`. `a` and `b` are substantially the same, and can be abstracted into `c`.

```
(define c
  (lambda (time name)
    (println "I am running: " name)
    (callback (+ time 40000) c
              (+ time 40000) name))) ;; arguments time and name

(let ((time (now)))
  (c time "a")
  (c (+ time 20000) "b"))
```

Listing 8: Encapsulated state

Listing 8 spawns two temporal recursions, both executing over `c`. An important and not immediately obvious benefit of temporal recursion is state encapsulation. Listing 8 starts two concurrent temporal recursions — one with a `name` value of "a" and the other with a `name` value of "b". `name` and `time` are maintained independently by their respective temporal recursions. Of course any number of independent temporal recursions over function `c` could be started (resources notwithstanding), each with its own independent state for `time` and `name`.

As well as independent state it is also possible to introduce shared state for temporal recursions. In Extempore closures provide the vehicle for introducing shared, but still encapsulated, state for temporal recursions. Consider a variation to function `c` that introduces a *captured* variable `count`.

```
(define c
  (let ((count 0))
    (lambda (time name)
      (println "I am running: " name " count: " count)
      (set! count (+ count 1))
      (callback (+ time 40000) c
                (+ time 40000) name)))) ;; time and name


(let ((time (now)))
  (c time "a")
  (c (+ time 20000) "b"))
```

Listing 9: Encapsulated shared state

In Listing 9 `count` is shared between both temporal recursions but is still encapsulated. Most importantly, because temporal recursions are non pre-emptive, access to the shared variable `count` is strictly ordered - temporally ordered. This makes shared temporal recursion state easy to reason about. Strict temporal ordering removes all but one ambiguity - the case of two temporal recursions having exactly the same scheduled time. In Extempore's case scheduled callbacks with identical scheduled times are called in the order in which they were added to the scheduler - FIFO.

There is no requirement for the `callback` time to be periodic. By adjusting the increment to `time` a temporal recursion can be aperiodic or sporadic, as demonstrated in Listing 10.

```
;; an example of aperiodic temporal recursion
;; random duration of 1000, 10000 or 100000 ticks
(define c
  (lambda (time name duration)
    (println "I am running: " name)
    (callback (+ time duration) c
              (+ time duration) name  ;; time and name
              (random '(1000 10000 100000)))))  ;; duration

(c (now) "a" 1000)
```

Listing 10: Aperiodic schedules

Extempore's temporal recursions support, not only the *start* time of a temporal recursion, but also the *deadline* of a temporal recursion. An execution deadline constraint can be added to any `callback` to provide an exception handling pathway for code that executes beyond its scheduled deadline. Listing 11 shows an example of a single temporal recursion that uses more than its allocated time.

```
(define d
  (lambda (time)
    (println "time lag: " (- (now) time))
    ;; waste some time
    (dotimes (i 400000) (* 1 2 3 4 5))
    (callback (+ time 1000) d
              (+ time 1000))))

(d (now))
```

Listing 11: Time lag!

In Listing 11 time lag continues to increase because the `dotimes` loop is consuming

more time than has been allocated (1000 ticks). One answer to this problem would be to modulate the callback rate to better manage the execution time.

```
(define d
  (lambda (time)
    (println "time lag: " (- (now) time))
    ;; waste some time
    (dotimes (i 5000) (* 1 2 3 4 5))
    ;; self regulate optimial callback time
    ;; by passing new revised time rather than static time.
    (callback (+ time 1000) d
              (+ time 1000 (- (now) time)))))

(d (now))
```

Listing 12: Time regulation

In Listing 12 the temporal recursion will self-regulate. If concurrency was the only consideration then this might be an acceptable option. However, temporal recursions are not only about providing concurrency but also temporal constraint. It is not enough to run many things, it is necessary to run many things at precisely scheduled times. So the above *self regulation* option is not an option for most real-time domains.

Instead of modulating the timing accuracy of a temporal recursion to support the performance profile of the hardware, a preferable solution would be to inform the programmer that the precise timing was not supported by the hardware profile - at least not with the current performance profile of the code.

Extempore supports this idea by providing an explicit timing deadline constraint for each temporal recursion. An optional argument to callback provides a maximum execution time constraint after which an exception will be thrown to alert the programmer to

the temporal recursions inability to meet its execution deadlines.

```
(define d
  (lambda (time)
    (println "time lag: " (- (now) time))
    ;; waste some time
    (dotimes (i 500) (* 1 2 3 4 5))
    ;; added execution contraint deadline of 900
    (callback (cons (+ time 1000) 900) d
              (+ time 1000))))

(d (now))
```

Listing 13: Time constraints

Adding an execution constraint of `900` in Listing 13 results in an exception hook being called if function `d` does not complete its execution in under 900 ticks. This style of temporal constraint ensures that code either (a) meets its deadlines or (b) fails gracefully.

Ideally the system would provide these temporal constraints (i.e. the 900 ticks) without making the programmer calculate the complex timing interrelationships between all possible temporal-recursions. In practice this is a non-trivial problem which is made all the more difficult in live programming scenarios where the overall behaviour of the system is completely run-time modifiable. This run-time modifiability makes static temporal analysis very challenging indeed, although is an area where Extempore can, and should improve!

Extempore's temporal recursion events may be Scheme closures, Scheme continuations, Scheme macros, XTLang closures, XTLang macros, or native C functions, making temporal recursions extremely flexible.

Finally, it is trivial to make the temporal recursion design pattern behave syn-

chronously. A "synchronous" sleep requires just a few lines of code. `sys:sleep`, uses `callback` for scheduling, as outlined in Listing 14.

```
(define *sys:toplevel-continuation* '())
(call/cc (lambda (k) (set! *sys:toplevel-continuation* k)))

(define sys:sleep
   (lambda (duration)
     (call/cc (lambda (cont)
                (callback (+ (now) duration) cont #t)
                (*sys:toplevel-continuation* 0)
                #t))))
```

Listing 14: sys:sleep - full implementation

Listing 14 uses a synchronous style, based on iteration and `sys:sleep`, rather than a recursion and a `callback`.

```
(define a
  (lambda ()
    (dotimes (i 5)
      (println "I am running a" i)
      (sys:sleep 40000))))

(define b
  (lambda ()
    (dotimes (i 5)
      (println "I am running b" i)
      (sys:sleep 40000))))

(let ((t (now)))
  (callback t a)
  (callback (+ t 20000) b))
```

Listing 15: Synchronous iterators?

Temporal recursion is based on the simple principle that timed events can be scheduled recursively. However, a suitably powerful implementation of temporal recursion (one supporting closures and continuations for example) supports a rich temporal design space enabling an extended set of practical time and concurrency tools.

4.4. XTLang

Extempore supports two language interfaces: Scheme (for high-level reasoning in a managed environment) and a new language, XTLang (for lower-level reasoning about system level concerns). XTLang has been designed and built as a major contribution of this thesis. In deployed applications of Extempore, both languages are often mixed together in the one live programming session. Live programming in Scheme, with calls into XT-

Lang, allows the programmer to use a higher-level managed environment that may be easier to reason about when the performance and accessibility of the full XTLang stack are not required. For situations where reasoning about the architecture of the machine becomes critical, the cyber-physical programmer can freely move to XTLang. As Scheme and XTLang are fully-capable general-purpose languages in their own right, the ability to freely move between them gives the cyber-physical programmer great flexibility in practice.

XTLang was designed to meet the demanding worst case execution time (WCET) profiles and direct hardware interfacing necessary to build a broad spectrum of cyber-physical systems. XTLang was built to achieve this without unduly sacrificing the interactive development experience common to Lisp programming environments — an experience that is central to cyber-physical programming.

XTLang strives to be not only efficient with regards to time and space, but also to be *reasonable* and *predictable* with regards to time and space. XTLang is not ultimately concerned with maximal throughput but is instead primarily concerned with being an extensible systems language providing reasonable WCET performance. To this end, XTLang is designed (a) to directly support low-level architecture primitives; (b) to compile directly to native code via LLVM[48]; (c) to manage memory manually using memory zones (regions/arenas); (d) to be statically typed; (e) to support generic code that is always unboxed and fully specialised (monomorphised) (f) to support native SIMD and MIMD parallelism explicitly; and (g) to support in-line assembly language.

4.5. XTLang and Scheme

XTLang takes its s-expression syntax from the Scheme language, presenting a symbolic expression syntax and block structure common to all members of the LISP family of

languages.

Listing 16 is taken unmodified from the classic Scheme language text Structure and Interpretation of Computer Programs (SICP)[2]. The XTLang variant in Listing 17 highlights the similarity to Scheme with only minor modifications including a switch of keywords from `define` to `bind-func`, and a library function name change from `remainder` to `rem`.

```
;; Scheme implementation of GCD from `classic' SICP
(define gcd
  (lambda (a b)
    (if (= b 0)
        a
        (gcd b (remainder a b)))))

(gcd 15 12) ;; => 3
```

Listing 16: Scheme implementation of GCD

```
(bind-func gcd
  (lambda (a b)
    (if (= b 0)
        a
        (gcd b (rem a b)))))

($ (gcd 15 12)) ;; => 3
```

Listing 17: XTLang implementation of GCD

Although Listings 16 and 17 look virtually identical this likeness is only skin deep. The syntactic similarity masks significant underlying semantic differences. Three principal differences between XTLang and Scheme that are not immediately decipherable from

the source code alone are (a) XTLang code is *not* garbage collected (b) XTLang code is *not* interpreted, but is instead compiled directly to native machine code and (c) the XTLang code example is statically typed.

A further distinction between Listings 16 and 17 is the trailing top-level call to test the `gcd` functions `(gcd 15 12)` vs `($ (gcd 15 12))`. As Extempore is a bilingual environment, supporting both Scheme and XTLang languages, it is possible to evaluate both gcd examples in the same Extempore process. However, because Scheme and XTLang have distinct name spaces, and memory spaces, top-level calls require additional annotation. Noting that Listings 16 and 17 both share the same name, a top-level call to `gcd` must distinguish between a Scheme call to `gcd` and an XTLang call to `gcd`.

By default all top-level calls in Extempore default to Scheme (this is end-user configurable), so evaluating `(gcd 15 12)` on-the-fly from the top-level, will call into the Scheme code in Listing 16. In order to call the XTLang Listing 17 variant, the call to `gcd` must be wrapped in the form `($ ...)`; which takes any valid XTLang expression and compiles the expression into native code. The newly compiled native code block is then immediately executed. In this way, top-level XTLang expressions are subject to the same static analysis as any other compiled XTLang function.

4.6. Type System

It is advantageous when building and modifying systems at run-time (when live programming!) that errors be caught as early as possible. This is particularly important for a live systems programming language where the luxury of offline testing is either greatly limited or simply unavailable.

For live programming systems the ability to "stop the world" for debugging purposes is usually impossible, and always undesirable. The ability to catch errors statically is ad-

vantageous as it removes a class of run-time errors that would otherwise require run-time management. Prior work[77] demonstrated how heuristics can be used to automatically manage certain Scheme run-time errors. A video demonstrating this style of dynamic heuristic intervention in practice is provided in Appendix D. However these heuristic interventions would ultimately be better avoided in the first place, and many such areas can be caught statically, *before* they become a run-time concern. In order to support greater static checking, XTLang includes a state type system.

XTLang's type system supports a standard array of primitive and aggregate types including 8bit, 32bit, 64bit and 128bit integers; 32bit and 64bit floating point types; tuples (transparent to C structs) and fixed length arrays. Additionally Extempore supports a number of non-standard C primitive and aggregate types including a boolean type, sum types, a primitive SIMD vector type, and first-class lexical closures. Finally XTLang supports indirect referencing via pointers to both primitive and aggregate types as well as supporting recursive named types (in the form of product and sum types).

Explicit type annotations and the constraints imposed by a static type system are often perceived as introducing unacceptable "cognitive dissonance" into the programming activity [35][56]. In the context of live programming these criticisms seem particularly damaging as the cognitive demands on the "live" programmer are already high. In an attempt to minimise the imposition of the type system, while maximising its benefits, XTLang supports bounded polymorphism (ad-hoc polymorphism with rank 1 parametric overloading) and partial type inference.

Pierce and Turner[67] suggest three primary goals for a partial type inference algorithm in the service of a HOT[5] (higher order, typed) programming language/programming

[5] A HOT programming style is "a style in which (1) the use of higher order functions and anonymous abstractions is encouraged; (2) polymorphic definitions are used freely and at a fairly fine grain (for individual function definitions rather than whole modules) and (3) pure data structures are used instead of mutable state whenever possible"[67, p.2]. XTLang is a very HOT language with respect

style:

1. To make fine-grained polymorphism tolerable, type arguments in applications of polymorphic functions must usually be inferred. However, it is acceptable to require annotations on the bound variables of top-level function definitions (since these provide useful documentation) and local function definitions (since these are relatively rare).

2. To make higher order programming convenient, it is helpful, though not absolutely necessary, to infer the type parameters to anonymous function definitions.

3. To support a mostly functional style (where the manipulation of pure data structures leads to many local variable bindings), local bindings should not normally require explicit annotations.

XTLang conforms to these three goals with type inference supported for top-level function definitions, anonymous function definitions, and local variable bindings. Overall, XTLang's type system, and inference system, have been heavily influenced by the functional languages ML and Haskell, in addition to C whose ABI XTLang conforms to.

The type `[i64,i64,i64]*` of `gcd` in Listing 17 is statically inferred by the XTLang compiler. In Listing 17 the type `[i64,i64,i64]*` is inferred because the integer literal '0' in the expression `(equals b 0)` defaults to a 64bit integer. This 64bit literal '0' is enough type information for the XTLang compiler to fully resolve the `gcd` function.

In XTLang, a function's type identifies the return type first, followed by any argument types. An XTLang function that takes a single 32bit integer argument and returns a double value would be type `[double,i32]*`. Functions in XTLang are references, which are specified, as in C, by an appended `*` for each level of indirection.

to points (1) and (2), although cooling with respect to point (3).

In Listing 18 the XTLang compiler infers the type `[i32,i32,i32]*` due to the explicit annotation of the literal `0:i32`.

```
(bind-func gcd
  (lambda (a b)
    (if (= b 0:i32)
        a
        (gcd b (rem a b)))))

($ (gcd 15:i32 12))
```

Listing 18: GCD annotated with a 32bit integer literal

After evaluating both Listing 17 and Listing 18 the Extempore run-time will include *two* XTLang specialisations of `gcd`, one for `[i64,i64,i64]*` and a second for `[i32,i32,i32]*`. XTLang's `gcd` function has become an ad-hoc polymorphic function. Ad-hoc due to the fact that each polymorphic variant is derived from an independent code base, with independent (ad-hoc) behaviour.

Ad-hoc polymorphism in XTLang (function overloading) resolves using both the return type of the function, as well as any argument types. Failure to fully resolve the type of an adhoc-polymorphic function results in a static type error. An important consideration in Listing 18 is the top-level expression `($ (gcd 15:i32 12))` used to call `gcd`, which explicitly defines the integer literal 15 as 32bit. Without this explicit declaration the 64bit specialisation of `gcd` would be called instead of the 32bit specialisation, for in this context, and without explicit annotation, the integers 15 and 12 would both default to 64bit integers (`i64`). It is sufficient in this case to explicitly type only one of `gcd`'s two arguments.

Variable declarations in XTLang can be explicitly typed. Listing 19 provides explicit types for `a` and `b`, which alleviates the need to type the literal `0` as in example 18.

Recompiling (i.e. re-evaluating) `gcd` in example 19 replaces the existing `[i32,i32,i32]*` specialisation from 18. In this example the behaviour would be identical, but in Live Coding performances it is often desirable to have divergent behaviour from function overloading.

```
(bind-func gcd
  (lambda (a:i32 b:i32)
    (if (= b 0)
        a
        (gcd b (rem a b)))))

($ (gcd 15:i32 12))
```

Listing 19: GCD explicit 32bit arguments

4.7. Generics

Although the `gcd` code examples in Listings 17 and 19 share essentially the same function body, they are compiled into unique specialisations; Listing 17 into a specialisation for `[i64,i64,i64]*` and Listing 19 into a specialisation for `[i32,i32,i32]*`. These are distinct execution paths with divergent machine instructions (in this case 64bit vs 32bit paths). When defining overloaded functions it is inconvenient, and error prone, to be forced to replicate what is essentially the same *source code* for multiple specialisations. XTLang's polymorphic implementation is based around macro instantiation to fully specialised monomorphic code, commonly associated with C++ template programming and to the research of Kaes[43].

In XTLang's case, monomorphisation works by taking a generic code template and specialising (monomorphising) instance types on-demand. Unlike parametric polymorphism in functional languages such as Haskell and ML generic code templates in XTLang

are not values, but instead exist as generic templates in memory. The XTLang compiler is responsible for reifying unique specialisations of a given generic template in order to satisfy valid call sites where an extant monomorphisation does not already exist – i.e. where a monomorphic code path satisfying the call site does not already exist.

A property of XTLang's monomorphisation that is important in XTLang's role as a systems language is that XTLang guarantees fully specialised code paths. While many languages support the specialisation of code paths through just-in-time optimisation (C#, Java, JavaScript and many more), they *do not guarantee* specialisation, making it more difficult for the developer to reason sensibly about the actual code running on the hardware at any point in time. XTLang removes this uncertainty by guaranteeing that **all** polymorphic instantiations are fully specialised. Full monomorphisation at the point of instantiation enables direct interfacing with C, provides a pragmatic solution to non-uniform data sizes, and ensures that polymorphic code is easy to reason about from a low-level architectural perspective.

Listing 20 demonstrates a generic definition for `gcd` annotated with the type `[!a,!a,!a]*`, a closure of two arguments where the argument types and the return type are defined as *type variables*, which can potentially assume any type. However, `!a` must be consistent throughout the type definition, such that any specialisation of `gcd`, as defined in Listing 20, will have a return type that matches both argument types. Valid examples would include `[i64,i64,i64]*`, `[float,float,float]*`, `[i8,i8,i8]*` etc.; but not `[float,double,double]*`, which would be rejected as `!a` would not be consistent throughout the type.

```
(bind-func gcd:[!a,!a,!a]*
  (lambda (a b)
    (if (= b 0)
        a
        (gcd b (rem a b)))))
```

Listing 20: Parametric polymorphic GCD

`gcd`, as defined in Listing 20 is now available as a polymorphic call site, but is not itself a value. Indeed, at this stage, `gcd` in Listing 20 has not been statically compiled in **any** sense, and exists only as an abstract representation – essentially as a macro, as a code template. The generic template definition of `gcd` only takes on a reified form once one or more specific type instances are requested, type-checked, and successfully reified into unique ad-hoc specialisations. This behaviour is similar to C++ template programming.

The earlier live programming session introduced two ad-hoc specialisations for `gcd`, `[i64,i64,i64]*` from Listing 17 and `[i32,i32,i32]*` from Listing 19. Compiling the code in Listing 21 will result in one new specialisation being constructed for `gcd`, `[double,double,double]*`, with the extant ad-hoc specialisation for `[i64,i64,i64]*` persisting. XTLang will not override an explicit ad-hoc specialisation with a specialisation derived from a generic template definition. Additionally a user defined specialisation can always be added to explicitly override a generic specialisation. In the context of a *new* Extempore session, without pre-existing ad-hoc specialisations for `gcd`, two specialisations would be generated by compiling `test_gcd` (as there would be no pre-existing ad-hoc specialisation for `[i64,i64,i64]*`).

```
(bind-func test_gcd
  (lambda ()
    (* (gcd 3 2)          ;; integer spec
       (gcd 3.0 2.0))))  ;; float spec
```

Listing 21: Test of parametric polymorphic GCD

A complication raised by template programming is that the generic template itself is not an independent compilation unit – there is no machine code generated when you evaluate a generic template in XTLang. Until a particular monomorphism is required, at which point the generic template is reified into source code of a specific type, there are no guarantees about a template's soundness for a given instance type (or indeed for any type!). This unfortunately separates the template's design from its use, making life more difficult for both the library designer, who must consider the many potential uses of the template definition, and the end-user who might (and probably will) attempt to use the template in ways unimaginable to the template's author.

As an example of the problem consider the case of a `String*` type being applied at the call site to `gcd` as defined in Listing 20. In XTLang `=` is undefined for `String*`, and `rem` is also undefined for `String*`. An instantiation of Listing 20 for `String*` will clearly fail to compile – this is proper and as expected. The problem is that any errors raised by the template instantiation are likely to be removed from the expectation of the programmer. In this particular case the exact error raised by the XTLang compiler for the call `($ (gcd '15' '12'))` is: "Failed to find suitable generic match for call to rem arity(2) for type (_ String* String*)". The problem here is that the end-user should not need to know about `rem` and its type requirements. This is a relatively simple case but the situation becomes exponentially worse the deeper into the call stack the eventual type error is found.

XTLang attempts to address these concerns by supporting type constraints, similar to C++ Concepts and tangentially aligned to Haskell Type Classes[10]. Using XTLang's type constraint system a library designer can constrain the range of acceptable types that a generic function may accept. In the case of Listing 20, to integer and floating point types. Listing 22 shows how `gcd` can be constrained to numeric types only. Only one argument need be constrained as the type `[!a,!a,!a]*` ensures that all arguments (and return) must be alike.

```
(bind-func gcd:[!a,!a,!a]* -> (lambda (r a b) (t:number? a))
  (lambda (a b)
    (if (= b 0)
        a
        (gcd b (rem a b)))))
```

Listing 22: Constrained polymorphic GCD

Trying again to instantiate `gcd` with `($ (gcd '15' '12'))` results in the error "Failed to find suitable generic match for call to gcd arity(2) for type (_ String* String*)" with the significant difference being that the type error has been caught on the call to `gcd` rather than `rem`, due to a violation of the `gcd` type constraint on `(t:number? a)`.

4.8. User Defined Types

XTLang supports product types in the form of n-tuples.

```
;; simple_tuple returns type <double,i64,String*>
(bind-func simple_tuple
  (lambda ()
    (tuple 1.0 2 "three")))
```

Listing 23: Product Type

Composite end-user types can be defined as named n-tuple's with global extent. `(bind-type 3DPoint <f32,f32,f32>)` defines a new type ($f32 \times f32 \times f32$). Named tuples are automatically provided with a default data constructor of the same name.

```
(bind-type 3DPoint <f32,f32,f32>)

(bind-func scale_3dPoint
  (lambda (s p:3DPoint*)
    (3DPoint (* s (tref p 0))
             (* s (tref p 1))
             (* s (tref p 2)))))
```

Listing 24: Named type constructor

Listing 24 `scale_3dPoint` accepts two arguments, a scalar `s`, along with a `3DPoint*`. `scale_3dPoint` constructs a new `3DPoint` which is `p` scaled by `s`. Each dimension of the named tuple is referenced with `tref`. In this example, `s`'s type of `f32` is inferred. The constructor `3DPoint` returns a pointer to the newly constructed `3DPoint`.

An n-tuple may contain any valid type, and named types may be recursively defined.

```
(bind-type PointList <3DPoint*,PointList*>)

(bind-func list_of_three
  (lambda (a b c)
    (let ((l:PointList* null))
      (tfill! l a l)
      (tfill! l b l)
      (tfill! l c l)
      l)))
```

Listing 25: Defining a recursive type

Listing 25 shows the definition of a recursive type `PointList` and one example of how

a three element `PointList` might be constructed and returned, using `tfill!` (tuple fill), which takes a tuple, followed by 'n' tuple elements.

Additionally, each individual element in a named type may optionally be provided with a name. Named elements are provided with default accessor functions (getter and setter) of the given name.

```
(bind-type Person <firstname:String*,lastname:String*,age:i32>)

(bind-func print:[void,Person*]*
  (lambda (obj)
    (println "Firstname:" (firstname obj)
    (println "Surname  :" (lastname obj))
    (println "Age      :" (age obj))
    void)))

(bind-func test
  (lambda ()
    (let ((body (Person "John" "Doe" 32)))
      (age body 42) ;; setter
      (println "Surname:" (surname body) "age:" (age body)) ;; getters
      void)))
```

Listing 26: Defining a type with named elements

At run-time, XTLang's tuples are bit for bit compatible with C-ABI structs. This greatly helps in supporting toll-free bridging with extant C library code.

XTLang named tuples support type variables as shown in Listing 27.

```
(bind-type Pair <!a,!b>)

;; three pairs returns Pair{Pair{f64,f64}*,Pair{String*,i64}*}*
(bind-func three_pairs
  (lambda ()
    (Pair (Pair 1.0 2.0) (Pair '3' 4))))
```

Listing 27: Defining a recursive type

XTLang also supports sum types as demonstrated in Listing 28. Deconstructors for Extempore's sum types begin with a `$`. In `($Leaf t (n) 1 0))))`, `(n)` is a pattern followed by expressions for both success and failure. An extended example of Extempore's sum types is provided in Appendix B which presents Extempore's core library implementation of monadic List.

```
(bind-data Tree{!a} (Empty)
                    (Leaf !a)
                    (Node Tree{!a}* Tree{!a}*))

;; find the depth of a tree
(bind-func depth:[i64,Tree{!a}*]*
  (lambda (atree)
    ($Node atree (l r)
           (+ 1 (max (depth l) (depth r)))
           ($Leaf atree (n) 1 0))))
```

Listing 28: Algebraic Data Types

4.9. Memory Management

Efficient and temporally deterministic memory management is critical for the correct functioning of real-time systems and other performance critical domains. "Managed" memory languages are still a rarity in computationally intensive domains, particularly domains with real-time demands. Although a great deal of energy has gone into the research and production of incremental, concurrent and real-time garbage collectors, it remains the case that the time and space overheads inherent in these collectors are rarely compatible with computationally intensive real-time domains.

This presents something of a difficulty for the design of a full-stack live programming language. Ideally a live programming environment should protect the programmer from the kinds of memory protection faults common to the systems languages C/C++. On the other hand, a full-stack live programming environment aims to give the programmer explicit control over resource management.

4.9.1. The Stack and the Heap

XTLang forgoes garbage collection [6], instead favouring explicit memory management, allowing programmers to precisely tune their memory model for a given problem. The granularity, and regularity, of memory allocation/de-allocation can be shifted up and down - from a total pre-allocation of memory, through to extremely fine grained allocation and release.

The stack is used extensively in XTLang, and is also important in supporting XTLang's direct interoperability with C. XTLang encourages a style of value level programming that is well suited to a stack based memory discipline. In the preceding XTLang `gcd`

[6] Although Extempore as a whole does not. Extempore's Scheme language interpreter incorporates a concurrent, incremental, real-time garbage collector based on a four-colour variant of Baker [6] [77]

examples, all of the memory is stack allocated. As well as implicit stack allocation, it is also possible in XTLang to make explicit stack allocations using `salloc` (stack alloc), although this is uncommon in practice. The `let` form in XTLang, as in standard Scheme, binds one or more values to one or more symbols. In Listing 29 `let` binds the symbols `a` through `h` to stack allocated memory, either implicitly (as in `a`, `b` and `c`) or explicitly (as in `d` through `h`).

```
(bind-func test_salloc
  (lambda ()
    (let ((a 0)                          ;; stack allocate 8 bytes
          (b 0.0)                        ;; stack allocate 8 bytes
          (c:float 0.0)                  ;; stack allocate 4 bytes
          (d:double* (salloc))           ;; stack allocate 8 bytes
          (e:double* (salloc 4))         ;; stack allocate 32 bytes
          (f:i32* (salloc 4))            ;; stack allocate 16 bytes
          (g:|4,float|* (salloc))        ;; stack allocate 16 bytes
          (h:<double,double>* (salloc))) ;; stack allocate 16 bytes
      void)))
```

Listing 29: Stack allocation

XTLang also supports heap memory allocation. The following example shows explicit heap memory management through the allocation (and de-allocation) of an array of 10 64bit integers into heap memory.

```
(bind-func heap_mem_example
  (lambda ()
    ;; heap allocate 'arr' (80 bytes)
    (let ((arr:|10,i64|* (halloc)))
      ;; zero out memory
      (doloop (i 10) (aset! arr i 0))
      ;; deallocate
      (free arr)
      void)))
```

Listing 30: Allocate and de-allocate heap memory

In XTLang `halloc` is simply an aligned malloc. The XTLang compiler determines the allocation size based on the type of `arr`, which in this case is explicitly annotated as an array of 10 64bit integers. An array in XTLang is defined as `|num,type|`, where `num` must be a literal integer, and `type` is any valid XTLang type. Array access in XTLang is bounds checked at run-time, but pointer arithmetic is not (such that pointers provide a more dangerous high-performance alternative). A literal array in XTLang is `(array 1 2 3 4)`, a literal tuple `(tuple 1.0 1 "one")` and a literal vector `(vector 1.0 2.0 3.0 4.0)`.

Although XTLang supports `halloc` (aligned malloc) and `free`, this style of memory allocation is rarely used and is strongly discouraged in practice.

4.9.2. Zone Memory

An extension of the stack-based memory management provides an interesting middle ground between explicit un-managed management, and automatic managed memory. Although stack memory management is very often considered to be "un-managed" it is in fact a very simple memory management model.

In the common case the stack discipline is managed at the function call level. The stack grows with each new function call, with new allocations "pushed" onto the stack (i.e. growing the stack), and then symmetrically released, or "popped", from the stack when the call returns - the stack frame[97]. Function arguments, and return values, may also be handled simply and elegantly using the stack discipline [7]. The stack provides a memory management system where data is added and removed in a last-in-first-out fashion. Stack based memory allocation is simple and efficient, and is often directly supported by hardware. As well as providing an efficient execution model, stack allocation provides a simple programming model, making it straightforward for programmers to reason about, from a time and space perspective.

Unfortunately stack-based memory is also temporally and spatially inflexible. Data is either relatively short lived (the duration of the function call), or alternatively must be copied, incurring a substantial performance overhead. In practice stack memory is also often quite limited in size.

The temporal and spatial limitations of the stack are highlighted in the following example, which attempts to return a higher order function (HOF), with the `a1` variable captured by the closure.

```
(bind-func stack_example
  (lambda ()
    ;; stack allocate array
    (let ((a1 (array 1.0 2.0 3.0 4.0)))
      ;; return higher order function
      (lambda (a2)
        (* a1 a2)))))
```

Listing 31: Stack with HOF

[7] In practice, arguments and return values are passed by register where available.

Listing 31 highlights a common problem with stack based memory management for languages supporting higher order functions. `a1`, a closed variable in the lambda returned by `stack_example`, is stack allocated, and will be released as soon as `stack_example` returns. The lambda *returned* from `stack_example` is presumably designed to be called at some future time - a future time when `a1` will no longer point to valid memory. Furthermore, memory must also be allocated for the higher order function that is to be returned (a reference to its environment for example).

One solution is to extend the stack discipline to life-cycles beyond a single function call by introducing the concept of a memory zone. Zone's are stored in a stack of zones where each new Zone is pushed onto the stack as it is created. New memory allocations are always made from the topmost zone of the zone-stack. When the topmost zone is popped from the zone-stack any memory allocated against that zone is automatically freed.

This style of zone stacking is a natural fit for zones of lexical extent. For example, consider an expression `(memzone size body)` which pushes a new zone of `size` pre-allocated bytes onto the zone stack. Any allocations requested in `body` are registered against the pre-allocated memory in new zone. When the execution of `body` is complete, the new zone is popped from the zone-stack and the memory of `size` bytes is immediately freed.

```
(bind-func zone_example
  (lambda ()
    (memzone 1024
      ;; zone allocate array
      (let ((a1 (array 1.0 2.0 3.0 4.0)))
        ;; return zone allocated closure
        (lambda (a2)
          (* a1 a2))))))
```

Listing 32: Memory Zone with closure

The function `zone_example` in Listing 32 pushes a new memory zone (1024 bytes) onto the zone-stack. Both `a1`, and the closure being returned, are allocated from the new zone. Unfortunately `zone_example` exhibits the same problematic behaviour as `stack_example` - the new zone is popped from the stack at its lexical extent (just before `zone_example` returns). The result being that just as with `stack_example` the returned closure, and/or its bound variables point to invalid memory locations. The `zone_example` does however improve on `stack_example` in two important aspects; firstly, it is now clear where the closure being returned is allocated from; secondly, a zone can be pre-allocated with any available system memory, circumventing common stack size limitations.

The additional temporal and spatial flexibility afforded by Zone memory allows the caller, rather than the callee to push the new memory zone. Memory allocations made in the callee will be made against the topmost zone, the zone pushed in the caller, and will therefore still be valid in the caller once the callee returns.

```
(bind-func zone_example
  (lambda ()
    ;; zone allocate array
    (let ((a1 (vector 1.0 2.0 3.0 4.0)))
      ;; return zone allocated closure
      (lambda (a2)
        (* a1 a2)))))

;; print the result of a1 * a2
(bind-func call_zone_example
  (lambda ()
    (memzone 1024
      (let ((f (zone_example))
            (a2 (vector 2.0 2.0 2.0 2.0)))
        (println "result:" (f a2))
        void))))
```

Listing 33: Memory Zone.

In Listing 33 `call_zone_example` pushes a new zone (1024 bytes) onto the zone-stack and then calls `zone_example`. Both `a1` and the returning closure are both allocated against the topmost zone (i.e. the one created by `call_zone_example`. The closure is returned from `zone_example` and bound to `f`. `a2` is then allocated into the topmost zone. `f` is then called with argument `a2` and the result of `(f a2)` (also a vector allocated against the topmost zone), is printed to the log. Finally the `(memzone 1024 ...)` lexical extent is reached and all of the memory allocated is cleared in a single free before `call_zone_example` returns.

4.9.3. Zones and Processes

Memory zones in XTLang are auto-expanding. If allocation reaches the end of the available zone memory, a new allocation (doubling the existing allocation) is made, and linked (a simple linked list) to the active zone. In this way a zone can expand indefinitely, however, a zone can be hard limited to restrict the possible maximum size of expansion.

Zone memory is efficient, straightforward to implement, and easy to reason about (time and space). The general principle of Zone memory is that a single contiguous block of memory is pre-allocated on the heap. Smaller allocations can then be made against the zone very efficiently simply by incrementing an offset into the pre-allocated memory. At some point the entire zone can be either; *reset*, in which case the offset is returned to 0, and memory is simply over-written; or *de-allocated*, in which case all allocations are freed in a single call.

```
(bind-func test
  (lambda ()
    ;; allocate into zone at top of zone stack
    (let ((d:|4,float|* (alloc)))
      (dotimes (i 4) (aset! d i 0))
      void)))

(bind-func three_mem_zones
  (lambda ()
    ;; allocated 16 bytes into active zone
    ;; where the active zone is the top zone on the zone stack
    (let ((a:|4,float|* (alloc))) ;; straight let (not letz)
      ;; push new memory zone onto zone stack
      ;; allocate 16 bytes from new zone
      (letz ((b:|4,float|* (alloc)))
        ;; calling test with new zone at top of stack
        (test)
        ;; push new memory zone onto zone stack
        ;; allocated 16 bytes from new zone
        (let ((c:|4,float|* (alloc)))
          (doloop (i 4) (aset! c i 0)) ;; zero out memory
          ) ;; close zone, release c
        ) ;; close zone, release b and d (from test call)
      )
    void)) ;; a is leaked into top level process zone

;; when calling from the top level
;; the program is at the top of the zone stack
;; which must be the default process zone
(three_mem_zones)
```

Listing 34: Three nested memory zones

`letz` has similar behaviour to a standard `let` with the important distinction that `letz` creates, and activates, a new memory zone at run-time. Any calls to `alloc` (not `halloc` or `salloc`) made within the lexical bounds of the `letz` will be allocated into the newly created *active* zone. The lexical extent of the `letz` defines the lifespan of the memory zone, and any allocations made into the zone will be immediately destroyed (freed) upon exiting the scope of the `letz`.

In Listing 34, a call to `three_mem_zones` creates two new memory zones, and pushes them onto the processes zone stack. This then accounts for three memory zones in total - the default (top of stack) process zone, and the two new zones pushed onto the stack in turn. Once `three_mem_zones` returns to the top-level only the top-level process memory zone will remain. Of note in this example is the memory leak into the process zone caused by var `a`, and the capture and release of `d` from the call to `test`.

It is often the case that a large amount of intermediate work, that should be released, has been used to calculate a result that should be retained. Any zone allocated object, which is returned (i.e. the last expression) from a lexical memory zone that created it, is copied from its current zone (the zone into which it was allocated), into its surrounding zone (the zone preceding it on the stack).

```
(bind-func make_matrix
  (lambda ()
    (letz ((d:|9,float|* (call_some_complicated_function)))
      d)))
```

Listing 35: Zone copying

In Listing 35 a call to `call_some_complicated_function` may make many internal zone allocations before returning a zone allocated matrix which is bound by `letz` to `d`. Although the memory zone created by the `letz` will be released after returning `d`, the

contents of `d` are copied into the zone stacks preceding zone, before that release occurs. This is a deep zone copy with references to zone allocated objects (from the active zone) also being zone copied. In this way, it is possible to release all of the "working" while maintaining the "result". Zone copying applies to all XTLang objects including closures.

XTLang's memory zones need not be lexical in scope, they may also be managed temporally. XTLang provides programmers with the ability to create, push, and pop zones manually. Memory zones are first class in XTLang and the programmer is given first class control over their life-cycle.

```
(bind-func manual_zones
  (lambda ()
    (let ((z (create_zone)))
      (push_zone z)
      (callsomething)  ;; call something with 'z' zone active
      (pop_zone z)     ;; deactivate z
      (destroy_zone z) ;; free 'z' memory
      void)))

(bind-func first_class_zones
  (lambda ()
    (let ((z (create_zone)))
      (push_zone z)
      ;; allocate d in z
      (let ((d:10,float|* (alloc)))
        (pop_zone z) ;; pop z but don't destroy z
        ;; return z (can be reactivated or destroyed later)
        ;; z is first class, can be stored,
        ;; passed as argument, returned etc..
        z))))
```

Listing 36: Manual zone management

Listing 36 provides some basic examples of first class zone management. Although not commonly required, this first-class status has some important use cases. Consider for example a digital signal processing callback function, that processes a buffer of audio on each and every callback - in theory, indefinitely.

```
(bind-func zone_dsp
  (let ((dataout:float* (alloc 128))    ;; 128 frames
        (z (create_zone (* 1024 1024))))
    (lambda (datain timein)
      (push_zone z)
      (do_audio_processing_work datain dataout)
      (pop_zone z)
      (reset_zone z) ;; resets zone 'head' to start
      ;; return dataout
      dataout)))
```

Listing 37: Zone reset

Listing 37 demonstrates an efficient mechanism for pre-allocation using zone memory. A memory zone is closed over by `zone_dsp`. Any allocations made during a callback (i.e. in `do_audio_process_work`) will be automatically reset at the end of the callback. Allocations into the zone are efficient, and potentially expensive operating systems calls (malloc and free) are entirely circumvented, as the only OS level memory allocation is made once, at zone creation time (upfront), and the zone's memory allocations are simply reused, each time that `zone_dsp` is called.

4.10. Closures

All user defined functions in XTLang are first class lexical closures. Lexical closures in XTLang may be anonymous (Listing 38), they may be bound locally via `let` (Listing 39)

or globally bound via `bind-func` (Listing 40). Function application is standard prefix notation, as per the Scheme language.

```
;; lambda abstraction and application
((lambda (x) (* x 2)) 3)
```

Listing 38: Procedure Abstraction and Application

```
;; f bound and applied
;; within let binding
(let ((f (lambda (x) (* x 2))))
  (f 3))
```

Listing 39: Local Binding

```
;; f bound at top-level
(bind-func f (lambda (x) (* x 2)))
;; top-level application
(f 3)
```

Listing 40: Global Binding

Scope aside, the major semantic difference between the Listings 40, 39 and 38, is the application of memory zones. Top-level XTLang closures (bound globally using `bind-func`), are all constructed with an associated top-level memory zone - one for each top-level closure. Once the closure has been compiled, a memory zone is allocated for that global symbol, and any closure-level-code is executed, against that new memory zone. In the following example `(alloc)` allocates `|10,i64|` storage into the memory zone associated with `my_closed_over_array`. Additionally `myarray` is initialized with `(doloop (i 10) (aset! myarray i 0))` before the `(lambda ...)` is bound to the global symbol `my_closed_over_array`.

```
;; myarray is stored into the zone
;; associated with my_closed_over_array
(bind-func my_closed_over_array
  ;; this is closure-level-code, outside of the lambda
  ;; and executed at compile-time
  (let ((myarray:|10,i64|* (alloc)))
    (doloop (i 10) (aset! myarray i 0))
    ;; from here on this is closure-body-code (i.e. run-time)
    (lambda (x idx)
      (aset! myarray idx x))))
```

Listing 41: Closure memory zone

Providing language support for top-level *closure* memory zones alleviates the gratuitous use of globally defined memory allocations. Consider the simple case of an oscillator. An oscillator, must support the idea of state - of phase. There are three obvious ways in which this state may be modelled.

```
;; global memory allocation of phase
(bind-val OSCPhase double 0.0)

;; use global memory
(bind-func oscillator
  (lambda (amp freq)
    (set! OSCPhase (+ OSCPhase (* (/ TWOPI SAMPLERATE) freq)))
    (* amp (cos OSCPhase))))
```

Listing 42: oscillator with global state

```
;; phase memory managed externally
(bind-func oscillator
  (lambda (phase amp freq)
    (pset! phase (+ phase (* (/ TWOPI SAMPLERATE) freq)))
    (* amp (cos (pref phase 0)))))
```

Listing 43: oscillator with phase passed by reference

```
;; phase memory allocated into oscillator's own memory zone
(bind-func oscillator
  (let ((phase 0.0))
    (lambda (amp freq)
      (set! phase (+ phase (* (/ TWOPI SAMPLERATE) freq)))
      (* amp (cos phase)))))
```

Listing 44: oscillator with encapsulated "zone" memory

Listing 44 encapsulates the phase memory, removing the need to reference external memory - as is the case with examples 42 and 43.

These top level memory zone allocations are useful. However, example 42 is still quite limited in application, supporting only a single active instance. Ideally it should be possible to construct as many independent oscillators as required. Extempore supports this through the use of higher order functions.

```
;; phase memory allocated into oscillator's own memory zone
(bind-func oscillator
  (lambda (phase 0.0)
    (lambda (amp freq)
      (set! phase (+ phase (* (/ TWOPI SAMPLERATE) freq)))
      (* amp (cos phase)))))
```

Listing 45: higher order oscillator with encapsulated memory

Example 45 demonstrates how oscillators can be constructed at run-time, each with its own independent `phase`. `Phase` in this example, being captured at run-time by the returned `lambda`. In closing these oscillator examples, it is important to note that the `phase` captured by the second `lambda` is allocated into the current run-time memory zone - not the top-level `oscillator` closure zone. Indeed the closure's construction (i.e. the `lambda`) is also constructed within that run-time zone. However, this memory zone, may in turn be another top-level closure zone. Consider the application of an oscillator in a simple audio signal processing procedure called from the audio device for each required audio sample. `time` is in frames (`n` frames per channel), `chan` is the current channel - 0 or 1 in the case of a stereo audio signal.

```
;; o1 and o2 allocated into dsp's closure zone
(bind-func dsp
  (let ((o1 (oscillator 0.0))
        (o2 (oscillator 0.0)))
    (lambda (time chan)
      (if (= chan 0)
          (o1 0.5 220.0)
          (o2 0.5 440.0)))))
```

Listing 46: An application of a higher order oscillator

In 46 `o1` and `o2` are allocated into the top-level closure memory zone owned by `dsp`. This elegantly encapsulates the lifespan of the two oscillators, with their own internal states, within the `dsp` audio callback routine - tying them to it's own lifespan.

4.11. Extensibility

Extensibility in XTLang takes two primary forms. Firstly, XTLang's C-ABI compatibility allows XTLang to trivially extend 'C' library code. Secondly, Extempore's support for polymorphism and lisp macros, provide a powerful environment for library developers.

While XTLang strives to provide the performance needed for full-stack systems level programming, there is still great value in allowing XTLang to interface directly to extant C libraries.

Equally important is the issue of trust. Many legacy codes have been used, and abused, over many years. During this period of use, trust develops, and certain libraries become de-facto standards. This is particularly true in the scientific community, where trust in results is necessarily predicated on trust in the code that produced those results. For these reasons it is highly valuable to support 'C' (and by extension Fortran 2003+), and XTLang goes out of its way to be C-ABI compatible.

Calling into a 'C' shared library is as simple as loading the library, and providing XTLang with the type of any required symbol.

```
;; open dynamic library
(bind-dylib kernel32 "Kernel32.dll")

;; an alias for HANDLE -> typedef PVOID HANDLE
(bind-alias HANDLE i8*)
;; an alias for DWORD -> typedef unsigned long DWORD
(bind-alias DWORD i32)

;; define windows process priority classes
(bind-val REALTIME_PRIORITY_CLASS     i32 #x00000100)
(bind-val HIGH_PRIORITY_CLASS         i32 #x00000080)
(bind-val IDLE_PRIORITY_CLASS         i32 #x00000040)
(bind-val ABOVE_NORMAL_PRIORITY_CLASS i32 #x00008000)
(bind-val BELOW_NORMAL_PRIORITY_CLASS i32 #x00004000)

;; get address 'GetCurrentProcess' symbol from kernel32
;; and provide XTLang with a TYPE for that symbol.
(bind-lib kernel32 GetCurrentProcess [HANDLE]*)

;; get address 'GetCurrentProcess' symbol from kernel32
;; and provide XTLang with a TYPE for that symbol.
(bind-lib kernel32 SetPriorityClass [i1,HANDLE,DWORD]*)

;; XTLang can now use win32 native calls
;; to set a priority for the current process
(bind-func set_realtime_priority
  (lambda (priority)
    (SetPriorityClass (GetCurrentProcess) prority)))

;; set real-time priority for current process
(set_realtime_priority REALTIME_PRIORITY_CLASS)
```

Listing 47: Loading a library

Listing 47 demonstrates how the dynamic library `Kernel32.dll`, a core operating system library on the Windows platform, can be loaded and bound at run-time to trivially extend XTLang's core capabilities in order to provide XTLang with the ability to control process execution priority. It is important to recognise that there is no bridging going on here. There is no translation of call types, data representations etc.. Instead XTLang simply calls GetCurrentProcess and SetPriorityClass as if they were standard XTLang calls - which in essence they are, as XTLang uses the standard C calling convention [8].

XTLang's product types are also compatible with C language structs. Listing 48 highlights compatibility between XTLang produce types and C structs.

[8] XTLang actually makes use of two different calling conventions. The standard C calling convention is used almost exclusively, except in the case of optimized tail recursion, where XTLang uses a calling convention known as FastCC.

```
;; open dynamic library
(bind-dylib libuser32 "User32.dll")

(bind-type MOUSEINPUT <i32,i32,DWORD,DWORD,DWORD,i64>)
(bind-type INPUT <DWORD,MOUSEINPUT>)
(bind-alias LPINPUT INPUT*)
(bind-alias UINT i32)
(bind-alias DWORD i32)

(bind-lib libuser32 SendInput [UINT,UINT,LPINPUT,i32]*)

;; XTLang code to inject mouse events
;; into the Windows event loop
;;
;; Note that XTLang's product type's INPUT and MOUSEINPUT
;; are compatible with the win32 call SendInput:
(bind-func mouse-injection
  (let ((evt:LPINPUT (alloc))
        (mouse_input (tref-ptr evt 1)))
    (tset! evt 0 0)
    (lambda (x:i32 y:i32 down_or_up:i32)
      (let ((md (cond ((< down_or_up 0) #x0002)
                      ((> down_or_up 0) #x0004)
                      (else #x0001))))
        (tfill! mouse_input x y #x0001 (+ md #x8000) 0 0)
        (SendInput 1 evt 40)))))
```

Listing 48: Loading a library

A common practice in Extempore is to work with an extant 'C' library, or application, and to slowly eat it from the inside out. This can be a very powerful tool, as it allows

an application developer to slowly introduce on-the-fly interactivity to an existing legacy code base.

For example, a C application, compiled with external symbols, can be loaded into Extempore, and its `main` routine can be replaced by XTLang code. Aside from this top-level replacement of `main`, all other code remains within the C executable. The XTLang replacement of main can then be executed inside Extempore - in a temporal recursion for example. Over time, as needs demand, more of the original C application can be replaced by XTLang code, without degrading the performance profile of the original code. A particle in cell physics simulation library is used to provide a real-world example of this style of carnivorous live programming in Chapter 5.

Although this thesis focuses on the benefits of *live* real-time programming it does not follow that this *liveness* should be seen as detracting from a programmers ability to write thoughtful, considered, code. While liveness is a wonderful asset for prototyping, it can also be a valuable and useful tool for code re-factoring, API design, bug fixing, performance tuning, etc.. However, Extempore can also be a productive language in completely off-line contexts. A useful *live* real-time language must also be a useful *static* off-line language.

Ultimately, without good library support, a live programming environment is of limited value. A core language will not provide the level of domain specialty required to manage complex real-time scenarios; there will be no music made in a concert where every unit-generator must be programmed from scratch. There will be no photographic "plates" recorded from a robotic telescope session where the plate solver must be developed from nothing. There will be no scientific findings from a computational exploration of a laser-plasma interaction where the particle in cell (PIC) simulation framework has to be developed on-the-fly from first principles. In order for these domains to be *available* to

the live programmer, a suitable level of abstraction must be in place ahead-of-time.

The application of a live language 5

5.1. Introduction

In this section a number of case studies demonstrating Extempore's practical application "in the wild" are explored. Each case study represents a non-trivial exploration of Extempore's application in a particular domain area. These case studies were developed solely by the author, with the exception of the 'High Performance Computing' case study - which was developed in collaboration with colleagues Dr Ben Swift, Dr Henry Gardner and Professor Viktor Decyk.

5.2. High Performance Computing

5.2.1. Introduction

Scientific computing is very often associated with high performance computing. Simulation code, often running on clusters with thousands of distributed nodes, is commonly used to model a wide range of scientific phenomena such as climate prediction, ocean science, materials science, astrophysics, genomics, vehicle dynamics, combustion systems, nuclear fusion, etc.. It is very often the case in these simulations that increased numerical performance translates directly into increased fidelity. As these simulated models increase in fidelity they also increase in "realism", helping to provide ever greater understanding of the real-world phenomena being artificially modelled.

Maximizing the fidelity of these scientific models, while minimizing the cost of their execution on large supercomputers, is one of the central concerns of high-performance computing. C/C++ and Fortran are the de-facto standards for this type of work, a

situation that has remained virtually unchanged for over forty years. And yet, while there has been little change in the development of *high-performance* computing codes for scientific modelling, there has been a considerable shift in the scientific computing community towards dynamic languages. Libraries such as NumPy and SciPy, provide good numerical performance by "hiding" C code behind a Python veneer. Unfortunately, in practice, the relatively good numerical performance provided by libraries like NumPy is lost when code "bubbles up" into the Python wrapper.

Ideally, it would be possible to match the convenience of Python, with the performance of C, such that the performance impact of "bubbling up" would be mitigated. In this ideal, the programmer could replace scientific C/Fortran codes in an interactive and opportunistic manner. Slowly cannibalizing extant scientific codes from the inside out.

This case study demonstrates how Extempore can be used to cannibalise an extant C particle-in-cell simulation code, in order to facilitate free and fluid steering, tuning, diagnostics, and most importantly, scientific discovery. The case study demonstrates that Extempore supports "bubble up" semantics, a desirable property for any live-coding system, without sacrificing the performance or flexibility inherent in the original C code.

5.2.2. Cannibalizing a Particle-in-Cell (PIC) code

Particle-in-Cell (PIC) simulation [23] is a common modelling technique used in plasma physics and engineering. A PIC code can be roughly broken down into three main phases; a deposit phase, where the charge of local particles is deposited onto a grid; a solve phase, where the electromagnetic field equations are solved to obtain an up to date field; and a push phase, which uses the newly calculated field to push the particles. These three phases are then run in a "top-level" simulation loop with an incremental time-step. A simplified overview of the PIC simulation loop (provided in C), is outlined in Listing 49.

```
int num_particles = 1e6;
int grid_size = 512;
/* initialise the particle and field data */
float* part = init_particles(num_particles);
float complex* field = init_fields(grid_size);
/* main simulation loop */
while (step < num_steps)
{
  deposit(part);      /* deposit charge on grid */
  solve(part, field); /* calculate field */
  push(part, field);  /* move particles */
  step++;             /* increment timestep */
}
```

Listing 49: PIC overview in C

The `deposit`, `solve` and `push` procedures are designed to function in parallel across a distributed high performance computing cluster. The distributed PIC library code called in the above example is from an open source implementation written by Viktor Decyk. Viktor has provided a range of PIC codes, which he refers to as skeleton codes, on his UCLA website[22]. These PIC codes, "bare bones, but fully functional"[22], provide a PIC implementation designed for three layer parallelism at the node, thread and vector levels. The particular skeleton code used in this case study makes direct use of both MPI and OpenMP.

This case study begins by loading an instance of Extempore on each node of a distributed HPC cluster. In this case study MPI (OpenMP's mpirun) is used to initialize each of these Extempore instances. Once initialized the C PIC library is loaded into each distributed Extempore instance, and the top-level C calls `deposit`, `solve` and `push` are bound using Extempore's `bind-lib` call. Once bound, Extempore is free to make

toll-free calls into these C library procedures. A top-level XTLang loop can then replace the extant C top-level loop in Listing 50.

```
(bind-val num_particles i32 1e6)
(bind-val grid_size i32 512)
;; initialise the particle and field data
(bind-val part float* (init_particles num_particles))
(bind-val field float* (init_fields grid_size))
;; main simulation loop
(while (< step num_steps)
  (deposit part)          ;; deposit charge on grid
  (solve part field)      ;; calculate field
  (push part field)       ;; move particles
  (set! step (+ step 1))) ;; increment timestep
```

Listing 50: PIC overview XTLang

In Listing 50 the calls to `deposit`, `solve` and `push` are made directly into the C PIC library from XTLang, with XTLang providing a simple top-level simulation loop. Although simple, this transformation makes possible the insertion of additional analysis routines into the simulation loop. Listing 51 adds `draw_particles` and `draw_field` procedures, which are pure XTLang procedures for drawing the individual particles and vector fields using XTLang's graphics libraries.

```
(bind-val num_particles i32 1e6)
(bind-val grid_size i32 512)
;; initialise the particle and field data
(bind-val part float* (init_particles num_particles))
(bind-val field float* (init_fields grid_size))
;; main simulation loop
(while (< step num_steps)
  (deposit part)
  (solve part field)
  (push part field)
  ;; add the ``XTLang'' visualisation routines
  (draw_particles part)
  (draw_field field)
  (set! step (+ step 1)))
```

Listing 51: Add visualisation to run loop

As well as adding passive procedures, such as visualization or analysis routines, it may be desirable to steer the running simulation itself. A more direct computational intervention would be the addition of an external electric field.

```
;; main simulation loop
(while (< step num_steps)
  (deposit part)
  (solve part field)
  (push part field)
  (draw_particles part)
  (draw_field field)
  ;; add an external electric field
  (doloop (i grid_size)
    (inc field (cos (* 2PI (/ i grid_size)))))
  (set! step (+ step 1)))
```

Listing 52: Add external electric field

Listing 52 demonstrates how an artificial external force can be trivially applied to the current `field` for each time step. The results of this type of direct computational manipulation are immediately visible in the graphics rendering of the field and particle data, providing a tight feedback loop that is required for effective computational steering.

As well as adding to the top-level simulation loop, it is possible to replace extant C PIC library procedures on-the-fly. Listing 53 introduces a replacement `deposit` procedure. This code, as with all of the previous examples, is sent directly to each computational node on the distributed cluster, and may be compiled for different OS and CPU architectures.

The code is distributed across the network, compiled on the node, and hot-swapped into the running simulation, becoming active on the next time-step. By replacing extant C PIC library procedures, like `deposit`, it becomes possible to dive down into the simulation, slowly replacing the old code base as and when required, either for maintenance, performance, or exploration and discovery - to slowly cannibalise an extant C library.

```
(bind-func deposit
  (lambda (part)
    ;; for each particle (cpp: charge per particle), deposit the charge
    ;; on grid q (1D, first-order)
    (doloop (j num_particles)
      (let ((x (pref part j))      ;; read particle x position
            (x0 (convert x i32))   ;; closest grid point below
            (x1 (+ x0 1)))         ;; closest grid point above
        ;; interpolate & deposit charge at lower grid point
        (inc q x0 (* cpp (convert x0)))
        ;; interpolate & deposit charge at upper grid point
        (inc q x1 (* cpp (- 1.0 (- (convert x1) x))))))))
```

Listing 53: Deposit charge on grid

These examples have been taken directly from the paper "Live Programming in Scientific Simulation", written with co-authors Ben Swift, Henry Gardner and Viktor Decyk. The paper is published in the Journal Supercomputing Frontiers and Innovations[85].

A demonstration video "live programming" this PIC simulator is provided in Appendix D.

5.2.3. Steering

The previous section provides an example of how Extempore can be used to steer a computational simulation over time. Although the example shown is necessarily simplified the basic components are evident. Computational Scientists are free to dive into sections of the code-base, as demand or interest arises, and make direct modifications to the running system. Furthermore these modifications can be made to a single node, or simultaneously across a cluster of nodes, at the programmer's discretion (or any sub-selection of nodes). The cluster may be homogeneous or heterogeneous, with respect to both Op-

erating System, and CPU architecture, to the extent that Extempore supports Linux, Windows and MacOS operating systems, as well as x86 and ARM CPU architectures.

A survey by Mattoso et al. [55] highlighted that domain-level interactivity was still a major open challenge in HPC workflows. While dynamic high-level HPC workflow management has improved, the area of dynamic domain interactivity has made little progress. Mattoso et al., highlight that adding new behaviours to a running system remains an open challenge. Cyber-physical programming in Extempore demonstrates the potential to address this challenge by making the full-stack of domain code available for run-time modification and extension, and by providing the ability to add completely novel, new domain code as required.

5.2.4. Rapid Prototyping

There is an understandable reluctance to experiment on supercomputing systems that are expensive to run, and oversubscribed. In this environment the costs associated with live programming experimentation may be considered too great. It is likely then, that live programming may be most at home on HPC workstations, in a rapid prototyping/development capacity. This style of HPC workstation development seems particularly attractive with new on-socket Intel PHI Knights Landing processors bringing truly high performance parallel computing closer to the desktop.

In this capacity live programming provides an excellent rapid prototyping tool, providing developers with the ability to move beyond iterative software development (Agile,XP), and into the realm of interactive development. While the idea of "fail fast, fail often" may seem like the antithesis of good engineering these ideas have found their way from enterprise computing into high performance computing, and are now widely regarded as constituting best practice [41].

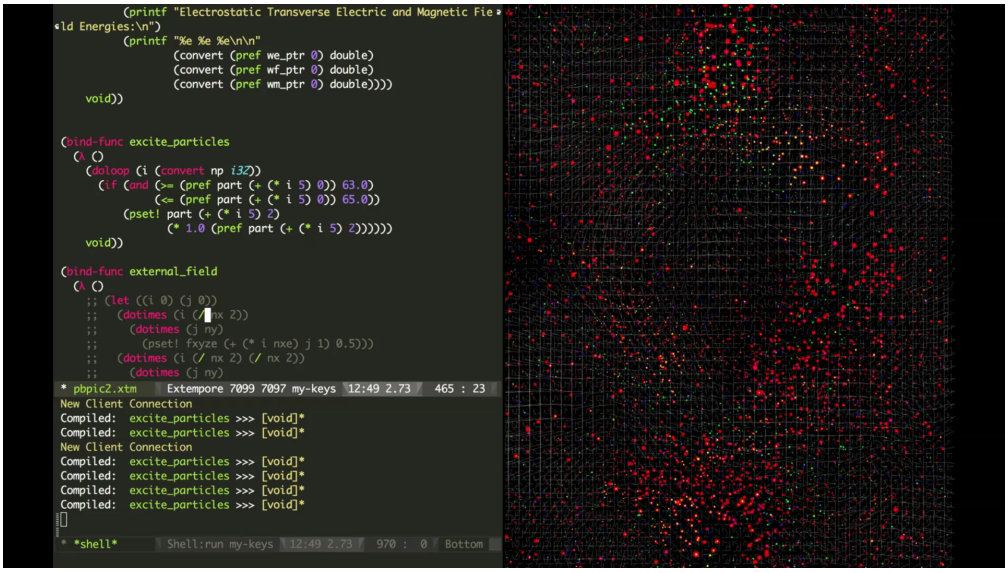


Figure 5.1.: Steering a PIC simulation in Extempore.

These early prototypes are often developed using higher level languages. They are very often serial implementations designed to ensure domain level correctness, rather than fully optimized, parallel implementations designed for high performance. Unfortunately this often means that these prototype environments have little in common with the final (usually completely rewritten) production codes.

Two problems here are disentangling production code from prototype code, and adequately testing the domain problem with vastly less efficient implementation code. In contrast to higher level languages, Extempore provides an effective live-coding environment suitable for rapid prototyping, but also provides the down-to-the-metal performance required to implement HPC production code. Extempore provides the ability to move smoothly from rapid prototyping into production with the same code base. Ideally the computational scientist will then be able to run exactly the same codes in an interactive exploratory fashion, on her local HPC workstation, and then to take those exact same codes, and run them on "big iron". Perhaps most importantly this process allows

the scientist to incrementally improve on previous results by live-coding on her local HPC workstation or local HPC cluster between runs on oversubscribed Supercomputing facilities - moving back and forth as necessary.

5.2.5. Resource Management

The SuperComputing landscape is becoming increasingly heterogeneous. Heterogeneous hardware running a heterogenous collection of Operating Systems, often on distributed clusters. These heterogenous environments naturally leak up the abstraction tree and ultimately result in additional complexity for site managers, system administrators, and software developers. Processor affinity, memory locality, vector register sizes, phi-cores and GPGPU, cache lines, node management, code scheduling, network synchronization and time management, are all daily concerns for HPC developers and administrators.

Although this describes a complex picture there is also an opportunity to provide solutions that are more tailored to meet the exact requirements of the job at hand. Working towards solutions that provide a more equitable distribution of "on-demand" scientific computing services. Live programming provides an opportunity to manage these heterogeneous resources more dynamically. Steering resources on-the-fly to utilize existing resources more efficiently and effectively. Live programming has the potential to make cluster management, process deployment, run-time-scheduling, real-time resource analysis and optimization manageable.

5.2.6. Visualisation and Analysis

Interactive development provides an excellent base for real-time data visualisation and analysis. The undoubted popularity of dynamic interactive languages such as R and Python in the scientific community is a testament to the fact that scientists value an

interactive dialogue with their data.

However, it is increasingly the case in "big science" that data analysis and visualisation are being restricted to online processing. The amount of data being produced is simply too large to be stored, and instead must be managed in real-time as it flows through the system. Data analysis then becomes a challenging real-time problem. Further complicating the analysis of big data it is increasingly the case that data cannot be moved, and analysis must instead be moved to the data. The ability to analyse, visualize and synthesize data on-the-node, is becoming an increasingly important requirement for big science applications. As one example, the Square Kilometre Array (SKA) is expected to produce in excess of 100 petaflops of data per second[24].

Extempore may provide one possible solution to this problem, by coupling an interactive just-in-time programming experience, with an efficient down-to-the-metal programming language designed from the ground up for real-time processing.

5.2.7. Exploration

The most exciting possibility afforded by live programming of scientific HPC codes is the opportunity for truly unique scientific discovery. Where live programming may make a vital contribution is in providing a more fertile field for domain level experimentation and exploration. This is "computer as crucible"[11].

> Like contemporary chemists - and before them the alchemists of old - who mix various substances together in a crucible and heat them to a high temperature to see what happens, today's experimental mathematicians put a hopefully potent mix of numbers formulas and algorithms into a computer in the hope that something of interest emerges [11, p. 1].

As Borwein and Devlin point out in their introduction to experimental mathematics,

it is the *unpublished explorations* of mathematicians rather than their *published logical proofs* that are the bread and butter of day-to-day mathematics.

> ... take a look at the private note-books of practically any of the mathematical greats and you will find page after page of trial-and-error experimentation (symbolic or numeric), exploratory calculations, guesses formulated, hypotheses examined ... [11, p. 2].

In an age where computation is so integral to scientific (as well as mathematical) discovery, a faster, more direct feedback cycle, between algorithmic implementation and run-time execution can only have a positive influence on the journey of discovery. Extempore's support for human-in-the-loop computational experimentation, where changes to a code's textual representation can be immediately integrated into a running system, accentuates the role of human perception in the discovery process. The programmer operates both above-the-loop and also within-the-loop, providing a unique opportunity to causally direct, as well as to directly perceive, changes in the physical environment.

5.3. A Computer Music "Language"

5.3.1. Introduction

Previous work on the Impromptu environment had highlighted the limitations of a live programming language, where *imperative* access to the "machine" was restricted. The artificial boundary imposed by Impromptu's Scheme language implementation ensured that the programmer's ability to engineer solutions capable of fully utilizing a given hardware platform were significantly curtailed. This imposed some severe restrictions on the types of computational work that could be achieved on-the-fly and this was particularly problematic for low-level real-time audio signal processing. The promise of a live

programming environment capable of supporting on-the-fly development and modification of low-level digital signal processing code, was a significant motivation for the work discussed in this thesis.

However, it was also obvious in the development of Impromptu that music composition is as broad as human endeavour. A music composition may rely on a neural-network, a galaxy formation algorithm, voice-recognition etc., and these algorithms may need to run in real-time on a cluster of heterogeneous machines, from phones to supercomputers, distributed around the globe. In short, the diversity of problems to which a "computer music language" may be turned is limited only by the composers imagination.

How then to strike a balance between low-level imperative access to hardware, with higher-level declarative interfaces for composers and sound-artists? This case study explores the implementation of "a computer music language" built, from the ground up, in XTLang. For the purposes of this thesis "CML" stands for a computer music language written in XTLang.

5.3.2. Background

There are now several live-coding environments that support "live" DSP programming to one degree or another. Some of the better known include Chuck, SuperCollider [1], LuaAV and Gibber, as well as the patching environments MaxMSP and PureData (PD).

For much of the history of computer music there has been a distinction between programming at the control layer and programming at the signal layer. Where, and what, that demarcation represents has been one of the defining characteristics of various computer music languages.

One of the contentions of this thesis, is that this demarcation is often *artificially*

[1] Derivative front-ends to SuperCollider, such as Overtone, may be collectively considered as "SuperCollider" in this section.

imposed by limitations of the language and run-time environment, rather than being explicitly *designed*. In particular, many programming languages targeting computer music applications lack the time and space determinism suitable for low-level audio signal processing. This forces an artificial demarcation between low-level and high-level constructs in the computer music language [2]. Often this results in a segregation into two languages, a high-level "computer music language" and a low-level audio signal processing language, most commonly C/C++.

The purpose then of a live systems language like XTLang, is not to remove the necessity for higher-order structure, but instead to make higher-order structure a continuous and malleable design space - a design *choice*. A continuous design space which appears desirable, not only from the perspective of "liveness", but also from the perspective of an artistic practice based on novelty and exploration.

5.3.3. Integrating the Audio Device

At the core of CML is a native (C-ABI) callback function that is ultimately responsible for filling a buffer of audio samples, required by the audio device for continuous playback; a monotonic synchronous function. This process is ultimately managed by the audio device's operating system driver.

The CML does not concern itself directly with the device driver, but instead operates at the operating system level, accepting whatever internal buffer management the operating system imposes [3].

[2] No distinction is made between "dedicated" computer music languages like Supercollider and ChucK, and "general" languages like Python, Scheme or Ruby

[3] Although XTLang is certainly capable of working at the level of the device driver - even supporting inline assembly language where required

```
;; load the CoreAudio dylib on-the-fly
(define coreaudio
  (sys:load-dylib
    "/System/Library/Frameworks/CoreAudio.framework/Versions/A/CoreAudio"))

;; define a few CoreAudio types
(bind-alias OSStatus i32)
(bind-alias AudioObjectID i32)
(bind-type SMPTETime <i16,i64,i32,i32,i32,i64,i64,i64,i64>)
(bind-type AudioBuffer <i32,i32,i8*>)
(bind-type ATS <double,i64,double,i64,SMPTETime,i32,i32>) ;; audio time stamp
(bind-type ABL <i32,AudioBuffer*>)                  ;; audio buffer list
(bind-alias AudioDeviceIOProc [OSStatus,AudioObjectID,ATS*,ABL*,ATS*,ABL*,ATS*,i8*]*)
(bind-alias AudioDeviceIOProcID AudioDeviceIOProc)

;; make AudioDeviceCreateIOProcID call available to XTLang (on-the-fly)
(bind-lib coreaudio AudioDeviceCreateIOProcID
          [OSStatus,AudioObjectID,AudioDeviceIOProc,i8*,AudioDeviceIOProcID*]*)
```

Listing 54: Setting up an OSX CoreAudio callback Part 1

```
;; a trivial ring modulation
(bind-func dsp
  (lambda (in:SAMPLE time:i64 chan:i64 dat:SAMPLE*)
    (if (> chan 1)
        0.0 ;; if greater than 2 channels return silence
        (* in (cos (* 2.0 3.141592 440.0 (/ 1.0 44100.0) (i64tof time)))))))

;; define an XTLang IOProc
(bind-func dsp_callback:AudioDeviceIOProc*
  (lambda (inDevice inNow inInputData inInputTime outOutputData inOutputTime inClientData)
    (letz ((i 0)
           (sampletime (dtoi64 (tref inOutputTime 0)))
           (buffer (tref outOutputData 1))
           (channels (tref buffer 0))
           (samples (/ (tref buffer 1) 4)) ;; buffer size in bytes
           (outbuf (cast (tref buffer 2) SAMPLE*))
           (inbuf (cast (tref (tref inInputData 1) 2) SAMPLE*)))
      (dotimes (i samples)
        (pset! outbuf i (dsp (pref inbuf i) (+ sampletime i) (% i channels) inbuf))))))

;; start_sound sets dsp_callback as the callback for CoreAudio
(bind-func start_sound
  (lambda ()
    ;; for brevity we omit audio device initialization
    ;; dsp_callback_native *is* dsp_callback as defined above
    (let ((theIOProcID:AudioDeviceIOProcID* null)
          (err (AudioDeviceCreateIOProcID deviceid dsp_callback_native null theIOProcID)))
      (if (<> err kAudioHardwareNoError)
          (println "Error setting up audio callback:" err)
          void))))
```

Listing 55: Setting up an OSX CoreAudio callback Part 2

Listings 54 and 55 outline a bare bones working example of how a `dsp` callback function can be established in XTLang from scratch, using the operating system's native infrastructure (in this case OSX). The `dsp` procedure provides the bedrock above which all other audio signal processing in the CML is managed. In Listing 55 the model is a pull style architecture, although a push style architecture could be implemented just as easily - with or without operating system support.

The audio device drives the callback signifying the device's readiness to consume and provide a new buffer of audio data (i.e. audio in and out). The size of these buffers is generally configurable when an audio stream is initialized. Buffer size largely determines the overall audio latency in the system. A buffer size of 512 frames (a single frame can contain `n` channels), where the system's overall sample-rate was `44100` (the standard rate for the compact disc), would result in an audio latency of `11ms`. Extempore's CML defaults to a buffer size of 128 frames, or `3ms`. The callback is driven from a high priority native operating system thread.

One peculiarity in Listing 54 that is worth highlighting is `dsp_callback_native`, which is passed to `AudioDeviceCreateIOProcID`. Procedures in Extempore, such as `dsp_callback`, are closures, and as such have semantics beyond a standard C function call. Specifically, an XTLang closure is both a C-ABI function, and its lexical environment, stored in aggregate. These native wrappers, which are automatically generated by the XTLang compiler, allow arbitrary C code to call XTLang closures; and this in turn allows XTLang closures to be used directly as C function pointers.

This section has outlined a base layer for all audio signal processing in the CML. A `dsp` function, with the type signature `[SAMPLE,SAMPLE,i64,i64,SAMPLE*]*` that defines a sample-by-sample, interleaved (audio channels) callback function whose reference is passed to a high priority operating system process responsible for managing interrupt

driven callbacks (ultimately) from the audio device.

5.3.4. On-the-fly DSP

In Listing 55, the `dsp` function defines a simple ring-modulator, effecting an arbitrary stereo audio input signal. Once `start_sound` is called, and the audio device driver begins requesting audio buffers, `dsp` will be kept busy indefinitely. However, as Extempore makes it trivial to compile and rebind on-the-fly, the behaviour of `dsp` can be changed at any time.

```
;; a convenient type alias for dsp
(bind-alias DSP [SAMPLE,SAMPLE,i64,i64,SAMPLE*]*)

;; playback a 440hz sinusoid with amplitude 0.1
;;
;; note that as soon as dsp is compiled it replaces
;; any extant dsp and we hear an immediate audible change
(bind-func dsp:DSP
  (lambda (in time chan dat)
    (if (> chan 1)
        0.0
        (cos (* 2.0 3.141592 220.0 (/ time 44100.0))))))
```

Listing 56: DSP on-the-fly

Listing 56 introduces a small modification to 55, changing the input ring-modulator, into a simple sinusoidal oscillator. The change from modulating the audio input signal, to producing a pure sine-tone is immediate once the new version of dsp is compiled and swapped in.

5.3.5. DSP "by the numbers"

Higher-level audio languages have typically focused on abstractions based around either a mathematical *signal* style approach or a physical *unit-generator* style approach - where the signal style is most commonly associated with functional programming paradigms, and the unit-generator style is associated with graph, or object based approaches.

Rather than adopting either of these traditional abstractions, the CML instead questions the necessity for any additional abstraction over an imperative "by the numbers" approach. Although in Extempore this style often looks very similar to the signal, functional language approach.

```
;; bipolar white noise with a 75% gain reduction
;; running across `n' channels
(bind-func dsp:DSP
  (lambda (in time chan dat)
    (* 0.25 (+ -1.0 (* 2.0 (random))))))
```

Listing 57: White noise with Gain

There are no *signals* or *unit-generators* involved in Listing 57; just the product of a constant with the result of a call to random (i.e. white noise). A simple sinusoid can be implemented using `cos`.

```
;; some handy constants
(bind-val SR i64 44100)
;; SAMPLE is an alias for float
(bind-val T SAMPLE (/ 1.0 44100.0))
(bind-val PI SAMPLE 3.1415926536)
(bind-val PI2 SAMPLE (* 2.0 PI))
(bind-val PI2T SAMPLE (* PI2 T))

;; sinusoid 220hz, 0.25 amp
(bind-func dsp:DSP
  (lambda (in time chan dat)
    (* 0.25 (cos (* PI2T 220.0 (convert time))))))
```

Listing 58: A sinusoid

Example 59 mixes the preceding examples into a single stereo example producing white noise, from the left channel, and a 220hz sinusoid from the right.

```
;; whitenoise     -> left
;; sinusoid 220hz -> right
(bind-func dsp:DSP
  (lambda (in time chan dat)
    (* 0.25
       (if (= chan 0)
           (random)
           (cos (* PI2T 220.0 (convert time)))))))
```

Listing 59: Noise from left, sinusoid from right

The imperative approach to signal generation does not preclude abstraction. The most ubiquitous abstraction in audio signal processing is the oscillator.

```
;; a higher order function
;; which returns an 'oscillator'
;; that maintains the phase state
(bind-func oscil
  (lambda (phase)
    (lambda (amp frq)
      (inc phase (* PI2T frq))
      (* amp (cos phase)))))

;; oscil 220hz -> left
;; oscil 330hz -> right
(bind-func dsp:DSP
  (let ((o1 (oscil 0.0))
        (o2 (oscil (* 0.5 PI))))
    (lambda (in time chan dat)
      (if (= chan 0)
          (o1 0.25 220.0)
          (o2 0.25 330.0)))))
```

Listing 60: Abstracting an Oscillator

Listing 60 draws together two XTLang features that highlight its utility for digital signal processing.

First, XTLang's first class support for closures provides a natural abstraction for encapsulating state in a DSP signal chain. Listing 60 `phase`, `o1` and `o2` are all captured stateful variables. Moreover, being first-class allows `oscil` to return new oscillators on demand (note that there is a phase offset between `o1` and `o2`).

Secondly, XTLang's support for memory zones provides a natural vehicle for managing object lifetime in the DSP domain. `o1` and `o2` (and by extension `phase` in `o1` and `o2`) are both zone allocated into the `dsp` top-level closure zone. The lifespan of `o1` and `o2` is tied

to the lifespan of `dsp` itself. A programmer initiated recompilation (i.e. live-coding), of `dsp` will result in `o1` and `o2` being de-allocated, although if the code remains unchanged, new `o1` and `o2` allocations will take their place.

5.3.6. Higher level structure

Just as the full complexity of biology cannot be trivially reduced to a few fundamental physical laws, so too music composition cannot simply be reduced to a summation of continuous signals. The following presents *one* possible way that higher-order structure may develop in XTLang from the bedrock `dsp` routine. This example develops a polyphonic FM synthesizer which will be controlled by a few simple higher level music theoretic constructs. The example is provided to demonstrate a minimal example of what is involved in developing high-order structure from a metaphorical blank slate.

The first task beyond establishing a basic signal from `dsp` is to initiate some external control over that signal. This first example uses XTLang's ability to directly manipulate a closure's environment externally.

```
;; core dsp 'bedrock'
(bind-func dsp:DSP
  (let ((amp 0.0)
        (frq 256.0)
        (o1 (oscil 0.0))
        (o2 (oscil (* 0.5 PIs))))
    (lambda (in time chan dat)
      (if (= chan 0)
          (o1 amp frq)
          (o2 amp frq)))))

;; play a sequence of notes
(begin
  (dotimes (i 24)
    ;; set 'note' pitch to 128+i*20 hz
    (dsp.frq (+ 128.0 (* i 20.0)))
    ;; set 'note' amplitude to 0.5
    (dsp.amp 0.5)
    ;; sleep for 5000 samples between 'notes'
    (sys:sleep 4410))
  ;; amp back to 0.0 on completion
  (dsp.amp 0.0))
```

Listing 61: "Playing" the DSP routine

`dsp` now includes the captured variables `amp` and `frq`. These two variables control the amplitude and the frequency of both oscillator `o1` and `o2`; both left and right channels simultaneously. By default `amp` is 0.0, which ensures that `dsp` is silent. Executing the `dotimes` in Listing 61 sets the frequency of both oscillators (`o1` and `o2`), using XTLang's dot syntax notation `(dsp.frq (+ 128.0 (* i 20.0)))`, in increments of `20hz`

from `128hz` up to `608hz`. `(dsp.amp 0.5)` sets the oscillators amplitudes to `0.5`, to start them sounding (by default `amp` is `0.0`, producing no sound). Finally `(sys:sleep 4410)` sleeps the loop for 5000 samples, which results in an upward monophonic sequence of 'notes' where the frequency changes ten times per second. Finally, after exiting from the dotimes (after playing all of the notes), `amp` is set back to `0.0`, silence.

This simple example works quite well. However, it is difficult to discern individual notes, as there is no dynamic contour, or envelope, associated with the volume of each individual note. A simple remedy is to introduce a windowing function. This will provide each note with a contour and separate a notes duration from its onset time. In other words, a windowing function will allow one note to finish sounding before the next note starts sounding. The extension to `dsp` is minor, simple function composition.

```
;; a hanning window
(bind-func window
  (lambda (time width)
    (if (< time width)
        (* 0.5 (- 1.0 (cos (* PI2 (/ (i64tof time) (i64tof width))))))
        0.0)))

;; core dsp 'bedrock'
(bind-func dsp:DSP
  (let ((t 0)
        (dur 2000) (amp 0.0) (frq 256.0)
        (o1 (oscil 0.0))
        (o2 (oscil (* 0.5 PIs))))
    (lambda (in time chan dat)
      (if (= chan 0) (inc t 1))
      (* (window t dur) (if (= chan 0)
                            (o1 amp frq)
                            (o2 amp frq))))))

;; play a sequence of notes
(begin
  (dotimes (i 24)
    ;; reset t to 0
    (dsp.t 0)
    ;; set a random duration for each note
    (dsp.dur (random 1000 4000))
    (dsp.frq (+ 128.0 (* i 20.0)))
    (dsp.amp 0.5)
    (sys:sleep 4410))
  (dsp.amp 0.0))
```

Listing 62: Music routine 2

Having introduced a simple *envelope*, the next step is to introduce polyphony. The approach taken here is to implement polyphony using higher-order closures. One closure per note up to a maximum of 20 notes of polyphony.

```
;; abstraction for a 'note'
(bind-func note
  (lambda (frq amp dur a:SAMPLE b:SAMPLE)
    (let ((o1 (oscil 0.0))
          (o2 (oscil 0.0))
          (t 0))
      (lambda (chan)
        (if (= 0 chan) (inc t 1))
        (* (window t dur)
           (if (= chan 0)
               (o1 amp frq)
               (o2 amp frq)))))))

;; core dsp 'bedrock'
;; 20 part polyphony
(bind-func dsp:DSP
  (let ((cnt:i64 0) (i:i64 0)
        (out 0.0)
        (n null)
        (notes:|20,[SAMPLE,i64]*|* (alloc))
        (play-note (lambda (frq amp dur a b)
                     (inc cnt 1)
                     (aset! notes (% cnt 20)
                            (note frq amp dur a b)))))
    (lambda (in time chan dat)
      (set! out 0.0)
      (dotimes (i 20)
        (set! n (aref notes i))
        (if (not (null? n))
            (inc out (n chan))))
      out)))
```

Listing 63: Music Routine 3

Taking advantage of XTLang's support for top-level closure zones, `dsp` introduces a cache for holding up to 20 active *note* closures - `notes`. Individual notes are added to the cache via the `play-note` closure. `play-note` spawns new notes by calling the higher-order closure `note`. It should be no surprise that `note` in this example, looks very similar to `dsp` in previous examples. `note` simply takes the monophonic dsp code from previous examples, and makes it available as a higher order function. Many monophonic notes can then be summed to produce a polyphonic output. This is then the task of `dsp`, to produce a polyphonic output by summing all of the active monophonic notes.

Finally, once polyphony is in place, the new note infrastructure can be used to provide a higher level interface for *playing* notes. Listing 64 introduces a `play` function, that accepts a pitch, volume, duration, as well as some optional synthesis parameters.

```
;; convert midi pitch numbers to frequencies in hz
(bind-func midi2frq
  (lambda (pitch:SAMPLE)
    (if (<= pitch 0.0) 0.0
        (* 440.0 (pow 2.0 (/ (- pitch 69.0) 12.0))))))

;; convenience for play
(bind-alias PN [[SAMPLE,i64]*,SAMPLE,SAMPLE,i64,SAMPLE,SAMPLE]*)

;; an abstraction for 'playing' a note
(bind-func play
  (lambda (pitch vol dur a b)
    (let ((f:PN (dsp.play-note)))
      (f (midi2frq pitch) (/ vol 127.0) dur a b))))
```

Listing 64: Music Routine 4

With all of this in place, performing melodies is now a straight-forward task. Listing

65 demonstrates a "high level" melody and accompaniment.

```
;; define two 'melodies'
(define melody (list 60 67 70 75))
(define accomp (list 36 48))

;; loop over two melodies
(define riff
  (lambda (idx)
    (sys:sleep (/ *second* 8))
    (play (rotate melody -1) 20.0 3000)
    (play (rotate accomp -1) 20.0 3000)
    (riff (+ idx 1))))

;; start playing
(riff 0)
```

Listing 65: Music Routine 5

Of primary importance is that everything that has been built can be changed live and on-the-fly in front of an audience. This last example 66 extends `note` with FM synthesis; extends `dsp` with a stereo delay effect; and extends `riff` by providing `play` with additional FM synthesis parameters.

```
;; frequency modulation synthesis
(bind-func note
  (lambda (frq amp dur a:SAMPLE b:SAMPLE)
    (let ((o1 (oscil 0.0))
          (o2 (oscil 0.0))
          (m1 (oscil 0.0))
          (m2 (oscil 0.0))
          (t 0))
      (lambda (chan)
        (if (= 0 chan) (set! t (+ t 1)))
        (* (window t dur)
           (if (= chan 0)
               (o1 amp
                   (+ frq
                      (m1 (* a b frq)
                          (* a frq))))
               (o2 amp
                   (+ frq
                      (m2 (* a b frq)
                          (* a frq)))))))))))
```

Listing 66: Music Routine 6

```
;; stereo delay effect

(bind-func dsp:DSP
  (let ((cnt:i64 0) (i:i64 0)
        (out 0.0) (decay 0.5) (idx 0)
        (dt:|2,i64|* (alloc))
        (dely:|88200,SAMPLE|* (alloc))
        (n null)
        (notes:|20,[SAMPLE,i64]*|* (alloc))
        (play-note (lambda (frq amp dur a b)
                     (inc cnt 1)
                     (aset! notes (% cnt 20)
                            (note frq amp dur a b)))))
    (afill! dt (/ SR 5) (/ SR 3))
    (lambda (in time chan dat)
      (set! out 0.0)
      (dotimes (i 20)
        (set! n (aref notes i))
        (if (not (null? n))
            (inc out (n chan))))
      (set! idx (+ (% time (aref dt chan)) (* SR chan)))
      (aset! dely idx (+ (* decay (aref dely idx)) out))
      (aref dely idx))))


(define riff
  (lambda (idx)
    (sys:sleep (/ *second* 8))
    (play (list-ref pitches (modulo idx (length pitches)))
          20.0 (/ *second* 12) 3.0 (* 5.0 (window (modulo idx 50) 50)))
    (play (list-ref pitches2 (modulo idx (length pitches2)))
          20.0 (/ *second* 12) 3.0 1.0)
    (riff (+ idx 1))))
```

Listing 67: Music Routine 7

Although this example may appear quite long, it is important to remember that everything here has been built from the ground up - from base operating system calls into a simple, but useful, computer music language complete with a polyphonic FM synthesizer. It is hoped that the reader will see just how easily this basic "computer music language" can be extended - at all levels, from signal through to control. Indeed, this simple example provides a rough analogue of Extempore's actual computer music engine — an engine that is used by the author in live-coding performances around the globe.

The example provided here is just one way in which a computer music language could be developed in Extempore. Extempore's actual DSP engine is significantly more advanced supporting a full spectrum of DSP processing routines and an efficient multi-core real-time engine. Appendix D includes one example from Extempore's core audio library as an example of a more sophisticated DSP effect; a vectorized partitioned convolution reverb.

5.4. The Physics Playroom

5.4.1. Introduction

The Physics Playroom was commissioned by the Queensland University of Technology as a launch application for their large multi-story interactive display wall, "The Cube"[70]. The Playroom is an exploratory interactive space designed to engage high school students and the general public with some of the fundamental ideas and personalities of the physical sciences. As one of only four launch applications, the Physics Playroom was expected to showcase the substantial monetary investment made in The Cube's physical infrastructure, as well as to provide an exemplar for future Cube projects. The development of the Physics Playroom was carried out over an eight month period between

April 2012 and November 2012 by the author and a single 3D graphic artist (Warwick Mellow). The Physics Playroom was developed entirely within Extempore.

Due to the high visibility of the Cube project, operational expectations were high. The launch applications were expected to provide 99.99% up-time, as well as delivering an engaging end-user experience across a wide target audience. In operation The Cube currently sees upwards of 75,000 unique visitors per week and has been visited by many luminaries.

Figure 5.2.: The author demonstrating Extempore and the Physics Playroom to Steve Wozniak

Despite the sensitivity of the project, there was some latitude for experimentation due in part to the university context, but perhaps more significantly due to the unique nature of The Cube's physical infrastructure. This unique physical infrastructure required bespoke solutions that encouraged a degree of experimentation and allowed the author to convince the selection committee to support a largely untried software development

methodology (cyber-physical programming) and a software development environment still in production (Extempore).

Video documentation of the Physics Playroom is provided in Appendix D.

5.4.2. Overview

The Playroom is a collaborative space that is designed to appear as a single unified scene. Human interactions in the space are designed to encourage cooperation by being inherently interconnected. A user may throw "a block" into another person's "building" even though they are working at different ends of the distributed display wall. This same general idea of interconnectedness pervades the Playroom and mandates that the Physics Playroom must be coordinated, as well as distributed.

The Playroom is displayed across twelve 55-inch multi-touch screens configured in a permanent wall mounted display. Additionally a stitched projection runs across the entire length of the twelve panel display effectively doubling the physical height of "the wall". The physical dimensions of the entire wall are approximately 17 meters wide by 3 meters high, while the pixel resolution is a combined 12960x3000 [4].

The 12 multi-touch panels and three stitched projectors run in high-definition 1080p. In order to render at this high pixel density the wall is comprised of seven render nodes - one render node per two touch displays (6*2 dual-head), with a final render node assigned to all three projectors (1*3 triple-head). These seven render nodes are not only responsible for rendering the unified scene but also for managing any other computational heavy lifting that may be required. For the Playroom these additional computational requirements were substantial - managing end-user interaction (touch, vision, sound), dynamic scene graph management, time synchronisation, physics simulation (classical

[4] Actual pixel density is slightly less than this combining both touch display and projection for a total of 12960x1080 + 5760x1920 pixels

mechanics and fluid dynamics) and audio signal processing.

The Physics Playroom supports an indefinite number of "cursors" (active fingers), although in practice was designed, and tested, to cope with 50 simultaneous users, each using all ten digits, without suffering noticeable performance degradation (i.e. 500 cursors). Fifty users is about the maximum number of "friendly" ten year old school children that can physically utilise the screen real-estate, although they certainly try to improve on that number!

The Physics Playroom incorporates a number of "physical simulations" into a fun and exploratory context. The intended goal of these simulations is to help engage lower high school students, and the general public, with the underlying mathematical models. The Playroom attempts to convey the importance and power of these general models by making them identifiable in an applied context - a 3D gaming world. The playroom concentrates on four particular models:

Newton's Laws of Motion The Physics Playroom is full of wooden blocks of a variety of sizes and shapes. All of these wooden blocks react to human touch (dragging, flicking, swiping etc.) and to collisions between themselves and other static objects in the scene. The blocks move in three dimensions and can be stacked vertically. Blocks behave according to standard classical mechanics allowing students to engage with basic concepts such as mass, acceleration and force.

Gravity All of the blocks in the Playroom are affected by Gravity. Gravity can be changed to reflect the particular characteristics of each Planet in the Solar system. Changing gravity "transports" the playroom to the surface of the chosen Planet. As well as planetary Gravity there is also a Zero Gravity option.

Fluid Dynamics The Playroom incorporates two large "paintings" which rotate through a

selection of famous portraits. These portraits are drawn using Extempore's particle system, where each particle is coloured to match a particular image pixel (or group of pixels) location. The movement of these particles is then driven by a fluid dynamics simulation which is in turn driven by human touch. The effect is to produce a type of sand painting, allowing users to perturb the "state" of the portrait.

Sound A series of sinusoidal oscillators are used to introduce the physics of Sound. Wave concepts including wavelength, amplitude and frequency are introduced in an intuitive and direct manner. These oscillators are chained to introduce the principles of amplitude and frequency modulation. The general public can manipulate the amplitude and frequency of each oscillator using a selection of dials. Sound transformations are heard in real-time and a real-time graphical representation of the audio signal is displayed.

5.4.3. Computational steering

A quality of live programming systems that is somewhat difficult to convey in text is that they are *always on*. When developing the Physics Playroom the system was always operational. The first edits of the morning have an immediate effect on the continuous evolution of the system. Software development in a live programming environment is building in the real-world - software development that takes place not only in the developer's computational environment, but also in connection to the physical environment.

One substantial benefit of an *always on* system is that the domain of investigation becomes *ready to hand*. This is a significant departure from the standard programming model where the task domain is usually only *present to hand*, available for study, for theory, but not for action[37][25].

By making the target domain of the programming activity ready-to-hand, live pro-

gramming emphasises a relationship with the domain which is both practical and instrumental as well as theoretical. From high level orchestration tasks, like synchronized rendering and distributed state, through to high performance compute tasks such as computational simulation, graphics rendering and audio signal processing, right down to the placement of individual objects and the lighting of the scene, live programming promotes the programming language interface as the dominant design tool, allowing the *activity* of programming to drive the design process.

Figure 5.3.: Australian Prime Minster Julia Gillard and Queensland State Premier Campbell Newman visit the Physics Playroom.

The always on quality of the programming experience encourages play as a valid and valuable way to resolve design problems. Often these decisions are made in the context of what feels right, such as manipulating the Playroom's fluid simulation to produce a

more "sandy like" behaviour. However, there are also many situations where an engineering problem can also be tackled more efficiently using a live programming dialogue with the environment. Successfully coordinating the activities of the many moving parts in the Playroom was an area where a cyber-physical programming practice proved to be extremely beneficial. With seven distributed rendering nodes responsible for the coordination of a single unified scene, there was a significant distributed computing component to the project. For performance reasons (display resolution), the scene needed to be distributed across seven machines. However, perhaps more significantly, the end-user interactions, and the *effects* of those interactions upon the computational simulation, were also distributed across the seven machines.

There is a degree to which live programming takes the increasingly short iterative development cycles embraced by the Agile and XP communities, and extends this idea to the point of continuity. Not to be confused with Continuous Integration (CI) - which encourages regular baseline merges, and automated system wide testing on-merge - live programming embraces an *always on* approach where software development occurs in a continuous and ongoing dialogue between the programmer, the program, the machine and the environment.

Consider the process of adapting the Playroom's bespoke fluid dynamics engine. Extempore allows on-the-fly changes to be made to the computational fluid engine while the simulation is running. Procedural modifications to the underlying computation can be made in direct response to direct gestural interactions with the touch screens. Our experience from this project was that *pushing and pulling* on the environment in this way made the development and tuning of the various physical simulations (mechanical, wave, sound, fluid etc.) considerably more efficient than would otherwise have been possible. When the system is *always on* the domain of investigation becomes immediately

ready-to-hand.

Figure 5.4.: Always on! Live programming the Physics Playroom in situ. This kind of live interaction between the live programmer, and an otherwise oblivious group of "users" turned out to be extremely valuable in practice.

A noticeable consequence of this physicality is that the ongoing computational model can be readily affected at anytime by external physical stimulus. This has the interesting consequence of embedding the programmer in a constant dialogue with the environment. There were many people moving through the Cube projects "skunk works" facility and they were often interacting with the Playroom, without the author being aware of it — as mentioned, the system is *always on*. Although this can occasionally be frustrating - when trying to track down a bug for example - these serendipitous end-user interactions lead to more effective and robust outcomes. There are far fewer surprises come launch day, if you have already had thousands of people using your application all-day every-day, while it is being developed.

5.4.4. The State of Things

The *always on* nature of the Playroom project focused attention on the *state of the system*, removing the focus of attention from the *state of the source code*. This is a positive outcome and an intended consequence of a live programming practice. However, the expectation of an *always on* system is that state is maintained indefinitely. In the event of catastrophic failure, or indeed of an orderly shutdown, the state of a live programming system should ideally be maintained such that a reasonable view of the system can be restored.

Unfortunately Extempore does not currently support a full restoration of state after system failure. Instead, the developer is expected to return to some canonical text representation of the code in order to "reboot" the system. This cognitive disconnect between the need to maintain an ongoing, and dynamic relationship with the computational system, and the necessity to maintain some notion of a canonical system state *in source code*, is a weakness in the current system and was the greatest source of tension while developing the Playroom. For while Extempore does maintain a text representation of all compiled code in memory, to be recalled and edited by the programmer on demand, this state is lost in the event of system failure. The programmer must therefore be diligent to maintain a canonical textual representation purely for purposes of "rebooting" the system in the event of a system shutdown/failure. This is clearly something that should be addressed moving forward.

5.4.5. Extempore for "product" development

Overall the ease with which high-performance simulation code could be modelled on-the-fly, reflects the benefits of incorporating XTLang into Extempore. This style of *malleable* high-performance computing would not have been impossible in the author's previous

Impromptu environment.

It is interesting to note that often this performance is of greatest use in resolving relatively trivial programming tasks - tasks which are cognitively undemanding and quite appropriate for sensorimotor coordination. The Playroom's particle systems provide a good case in point.

The Playroom incorporates two paintings which display the portraits of famous scientists and mathematicians. Each portrait is a particle system, whose particle's position, velocity, colour, size, lifespan, transparency etc. are parametrically controlled by the Playroom's fluid simulator and a bitmap of a particular "famous head". This relationship is parametrically complex but is not procedurally complex and XTLang makes this type of high-performance computation easy to develop, and perhaps more importantly to *steer* in real-time.

While it is certainly possible to provide the tools to enable less performant languages to control a particle system, these languages must ultimately pass off heavy computation tasks to a *hidden* software layer. This hidden layer is not simply a language divide but is more fundamentally an access divide. Real-time run-time changes below the divide are not possible. Working on the Physics Playroom indicated that in practice this matters.

Conclusions, reflections and future work 6

6.1. Reflections

There is something of a dichotomy inherent in the development of a cyber-physical programming system — the system's greatest strengths are also its greatest weaknesses. Liveness gives the programmer the ability to engage directly with the real-world on its own terms — within its own stream of natural events. And yet, liveness can be a distraction from the considered design, study and reflection that solving difficult problems requires. Full-stack programming provides an integrated solution encompassing the full software stack — no layer is too deep for the cyber-physical programmer! However, a reasonable separation of concerns into defined layers is a well understood engineering design principle. Do we really want application developers fiddling with the kernel? Certainly for the cyber-physical programmer, the ability to recode the numeric stack on-the-fly is powerful. Yet "recoding" the numeric stack of a "live" system may not be appreciated by all of those systems users!

How far should a system go to protect users from themselves? In the broadest sense Extempore chooses not to protect users from themselves where that choice would result in less power being afforded to the user. This does not entail that Extempore has been designed with no consideration to the end-user, but rather that where conflict arises between form and function, function has generally been favoured. Now that the project is reaching some basic level of maturity it is worth considering Extempore's usability more generally. How practical is Extempore for day-to-day software development?

Firstly, there does not appear to be any particular issue using Extempore for general purpose day-to-day "non-live" development purposes. In addition to the Extempore

environment itself, and to the case studies presented, Extempore includes a significant amount of non-trivial library code that has, to a large degree, been written "offline" — at least insofar as it was developed without any cyber-physical time constraints. The ability to work with Extempore "offline", following a more traditional programming paradigm, is not only beneficial, but probably crucial to Extempore's success. In short, there does not appear to be any problem supporting both "live" and "offline" modalities.

What can be problematic is moving *between* these two modalities. Session management is a significant limitation in Extempore's current design. At present, there is no temporal record of a programmer's live edits during a programming session, and the only representations available to the programmer at *any* point in time are (a) the live in-memory program state, and (b) any text available in the programmer's editor; which at best will only be a partial reflection of in-memory state; and at worst may have been completely deleted! This is clearly far from ideal.

In practice, session management is not as much of a problem as might be imagined. The author generally works with multiple text editor windows (text buffers), segregating the work into distinct workspaces, "live" and "offline": live workspaces, as a kind of scratchpad workspace, and offline workspaces for storing canonical code (i.e. for code destined to live beyond the session). While there is no tooling to enforce this segregation, and significantly the offline workspaces are still very much alive, the distinct separation of the text buffers helps the author to maintain a sensible ongoing text representation of the program (i.e. the *canonical* code).

When working on the case studies described in this thesis (and many other projects besides), it was clear that Extempore would benefit greatly from process imaging and remote database integration — both of these are core ideas from the Smalltalk community[45]. Extempore already includes an in-process database for storing source code,

source code documentation, and code analytics (such as static type information) and a good start would be to make this database available out of process. Additionally, the ability to store an Extempore session to disk, similar to Smalltalk "images" or to HPC checkpoint/restore methods, is extremely appealing but is yet to be investigated in detail.

Of course there is still the more fundamental question. Should programmers actually be allowed to modify the entire stack "on-the-fly"? [1]. Perhaps of even greater concern in the modern age are the security implications of a programming model where the full-stack is available for runtime end-user modification. These are difficult challenges that are beyond the immediate scope of the thesis but they are clearly provoked by Extempore's current implementation.

At a number of places in this thesis, XTLang has been described as being "reasonable" with regards to the understanding of computational time and space. It is important to clarify that being "more reasonable" does not in any direct sense make XTLang "easier" and it would be a mistake to infer this connection. To be able to *sensibly reason* about time and space from a performance perspective requires a sophisticated understanding of the machine's basic operation. This knowledge is usually beyond a beginner's level of understanding. Because of this, to speak of something being "more or less easy" in relation to being "more or less reasonable" is somewhat nonsensical, and at the very least needs to be heavily qualified — easier for whom? For the experienced practitioner who is writing low-level machine operations, the ability to write "reasonable" code in relation to the machine's underlying hardware is likely to be a boon. For novice users who do not yet fully understand the inner workings of the machine, XTLang's "reasonableness" may well make initial understanding more difficult.

The challenge then is to make the developer's life "easier" whilst also trying to be "rea-

[1] "Oops! I just typed 'PROCESSOR <- NIL", as the old Smalltalk joke goes[45, p. 23]

sonable". One way in which Extempore attempts this is via its dual language interface. The Scheme programming language, with its deep roots in education, provides a fantastic playground for novice programmers. Extempore provides a full R5RS[3] Scheme environment. Extempore's Scheme language environment has full access to Extempore's XTLang core libraries, giving "Scheme Extempore" programmers the ability to interact with Extempore's powerful audio, graphics, simulation and numerics libraries. This makes Scheme Extempore a powerful half-stack live programming environment in its own right. By providing a complete R5RS Scheme environment, basic "Scheme Extempore" can be taught and studied using any number of Scheme references[2].

In practice new Extempore users are often impatient to get their hands on XTLang. Everyone wants to jump in at the deep end! This causes something of a pedagogical problem as new Extempore users often do not know Scheme, and do not give themselves time to learn Scheme before moving onto XTLang. This complicates matters significantly as new users coming to Extempore are often learning, not one, but two new languages at the same time.

Extempore tries to make the boundary between XTLang and Scheme as easy to cross as practical. So much so that the two languages were designed to share largely the same syntax. However, the languages were intentionally designed not to share a semantics, and this can lead to problems with new Extempore users at the boundaries between XTLang and Scheme, where beginners are often likely to elide the two languages. It is worth pointing out that this is not directly about novice verses veteran programmers but is more fundamentally related to whether the new user comes to Extempore with prior knowledge of the Scheme language. It is difficult to know at this time whether this is a general problem with bilingual environments (particularly ones with two languages

[2]Such as the Abelson and Sussman classic "Structure and Interpretation of Computer Programs" [2]

sharing a similar syntax), or is simply that a considered pedagogy for Extempore has not been designed nor trialled. Anecdotally new Extempore users who come to the environment from Scheme (for example from the author's previous Impromptu project) have relatively little problem negotiating the XTLang/Scheme divide. This is not to say that they do not have learning difficulties with new XTLang concepts such as manual memory management, pointer manipulation and so forth, only to say that they appear to have little trouble identifying the boundaries between Scheme and XTLang. What appears to offer significant benefit is that new Extempore users with a background in Scheme have the opportunity to slowly immerse themselves into XTLang, whilst still being fully productive in a fully functional half stack live programming environment using Scheme.

As users begin to explore XTLang, they often struggle with the same types of learning difficulties common to beginner C programmers. For Scheme programmers, learning to work with XTLang's type system and direct memory management are the two greatest intellectual stumbling blocks. Anecdotally, what does appear to be a positive sign is that as Extempore users become increasingly familiar with XTLang they tend to prefer to work in XTLang almost exclusively. It is unclear if this is because of language features, such as XTLang's type system, or for the added performance and timing guarantees that XTLang provides. As Extempore users are generally a fairly pragmatic bunch, it is likely to be the latter.

Throughout this thesis the term "cyber-physical programming" has been used to describe the live programming of a cyber-physical system, and yet the term has never been precisely defined. This has been intentional, as the term "cyber-physical programming" is intended to suggest an aspirational practice of programming — one that is truly engaged with the real, physical world. The case studies have hopefully demonstrated that

Extempore does make a cyber-physical programming practice possible. The HPC case study was used for multiple live cyber-physical programming demonstrations (in front of an audience!) at Supercomputing 2015. These demonstrations utilised a distributed HPC cluster, incorporating "live" nodes in Singapore, Canberra (Australia), and Austin (USA). Extempore was used to provision the cluster (i.e. spin up nodes around the world), run a complex distributed Particle-in-Cell simulation, and finally to reduce and visualise the particle-in-cell simulation data being generated. All of these things were done with *live-edit* capability; with on-the-fly code changes working seamlessly across a widely distributed "global" cluster.

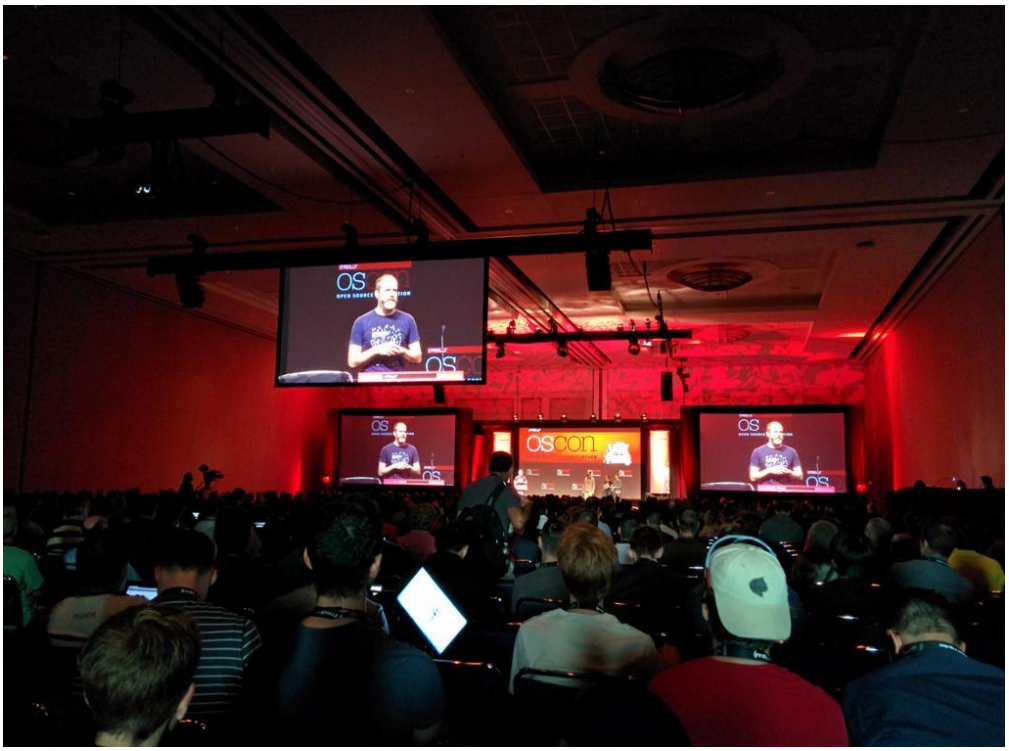


Figure 6.1.: Pressure cooker! The author live programming in front of 3000 people.

Cyber-Physical programming in Extempore is perhaps best known in the musical context. The author is well known for performing with Extempore all around the world

where his performances often attract audiences numbering in the thousands. Readers may reasonably question the safety of some of the design decisions made when building XTLang, such as manual memory management and direct pointer manipulation. Have these choices made Extempore an unsafe and unstable working environment? This is difficult to answer for the general case. But in the author's specific case, Extempore is used regularly in extremely high-stress and high-risk contexts, and in practice has proven to be remarkably resilient — when performing for 3000 people, through a million dollar sound system, you have to have some faith in the reliability of your tools!

At the time of its construction "The Cube" was the largest interactive display wall in the world. The Physics Playroom was launched in 2013, and ran continuously until late 2016. Extempore made a significant contribution to the success of the Physics Playroom, and, in particular, its very short 8 month development time. Significantly, this development time frame included building the *distributed* graphics engine, the physics engine, the sound engine, the animation library, the fluid simulation code, and all of the touch IO and inter-node communication and synchronisation. The Physics Playroom was marked by its high frame-rates and "silky smooth" touch interaction. Extempore's performance was notable, despite the fact that all of the other Cube projects were built upon commercial game engines including Unity, Unreal Engine, and Torqe3D.

What Extempore makes possible, particularly through XTLang, is the ability to engage with the real-world on its own terms — real-time, real-world "live programming" through the lens of a programming language interface. XTLang makes this possible by providing the programmer with good performance and low-level access to hardware, whilst maintaining the *live-edit* capabilities commonly associated with higher-level languages. Significantly, XTLang makes this possible when writing *idiomatic code*. Programmers should be able to reason sensibly about the time and space costs of their programs,

without having to resort to opaque, and non-idiomatic tricks and kludges.

However, while the thesis has outlined the potential of a cyber-physical programming practice — an aspirational ideal — it has been difficult to identify cyber-physical programming's "killer app". In some sense "the killer app" for cyber-physical programming is live-coding, i.e. musical live programming, which clearly needed no discovery! However, one of the initial goals for the thesis was to discover new domains "beyond live-coding" where cyber-physical programming could be clearly shown to provide superior outcomes.

It is unclear whether steering HPC simulations, such as the PIC codes described in this thesis, will actually lead to new insights and scientific discoveries. Live-coding a PIC simulation makes for great demonstrations, but does it lead to new discoveries? Possibly, but this is far from clear from the work presented here. What does appear to be valuable is a "modern" systems level language and an ability to work *across* stacks. Rapid interactive development of low-level systems codes has proven to be valuable. It may prove to be the case that strong integration and tight iteration are ultimately more valuable than the true liveness offered by Extempore. In other words, it may be the case that integration is ultimately more important for "liveness" than time. Time will tell!

6.2. Future work

6.2.1. Bootstrapping XTLang

Extempore's XTLang compiler is currently written in Scheme. At present, this means that compilation times for XTLang code are slower than ideal. In practice, for single shot code compilations, such as hot swapping a function on-the-fly, the compilation time is fast enough so as to be largely inconsequential. More problematic are Extempore's ever expanding core libraries. The Extempore project now hosts close to 100K lines of

library code, completely distinct from any core Extempore language code. Compiling these libraries takes considerable time (five minutes on a standard desktop). Extempore does support the 'offline' compilation of dynamic libraries, which does significantly reduce the run-time dependence on the XTLang compiler, but, nevertheless, the performance of the XTLang compiler is an area of concern. XTLang is now more than capable of being bootstrapped, and work is underway to replace the Scheme XTLang compiler with a self-hosted XTLang compiler.

6.2.2. Running on-the-metal and time predictable architectures

Extempore's design was (at least in part) motivated by two technical developments that have not yet been fully realised. The first was a desire to run Extempore directly on raw hardware (i.e. without a host operating system). The second (and related) motivation was to support a more robust time-predictable architecture. One of the original motivations for using LLVM, and to approaching live programming more generally from a low-level architecture perspective, was the hope of providing the programmer with far greater certainty about the execution profile of their compiled code; in particular the WCET (worst case execution time) of their compiled code.

As discussed in Chapters 3 and 4, *commodity operating systems* are not designed with cyber-physical programming in mind. In part, Extempore's solution to this problem is to be flexible — by being close to the hardware, Extempore programmers have the tools required to circumvent the operating system if required. While this may not always be possible it does provide programmers with the best possible choice. Exploiting low-level access is exactly what the programmer is doing when directly assigning cores, or accessing memory through a NUMA API; they are circumventing the *standard* operation of the OS. Ultimately in Extempore this extends to the CPU itself via inline assembly level

access to the CPU's native instruction set.

Unfortunately it is not only commodity operating systems that are incompatible with real-time systems programming, but also *commodity hardware*. Although modern super-scalar architectures make timing predictability difficult[72], there is a current trend in CPU design to embrace simpler, more deterministic architectures[44, 72]. One important future direction is to self-host (i.e running on the metal) Extempore on simpler, scratch-pad based, CPU architectures. This would provide greater opportunity to explore static timing analysis and scheduling within the context of live programming, and could be an exciting platform for cyber-physical systems development.

6.3. Conclusion

Cyber-physical programming gives programmers the ability to engage actively and proactively in an experimental and experiential relationship with complex virtual, physical and cultural systems. Cyber-physical programming offers programmers the opportunity to actively shape and control these complex systems of events — to be *engaged* with them. Studying these natural events is often only possible in-situ, in-the-world, within their own temporal and spatial frames of reference.

Cyber-physical programming posits that a causal connection to the world is valuable and that it can be useful to privilege the present. To quote Rodney Brooks "It turns out to be better to use the world as its own model."[15, p1]. One of cyber-physical programming's tenets is that a program's state is not only an internal property of the machine, but is also an external property-of-the-world. This external, physical state, can be sensed and acted upon by both the machine *and* the programmer. The programmer is actively engaged in both perceiving and acting upon the world, directly, but also mediated through the machine. For the cyber-physical programmer, the physical environment

constitutes a meaningful component of a program's state.

A system's ability to synthesise — to create behaviours by acting in the world — is what enables systems to function effectively in-the-world. For this synthesis to be possible the system must engage directly with the flow of real-time natural events that surround it. What Extempore ultimately strives to do is to keep up with the flow of natural events that are of concern, at a specific time and place, to a particular live programmer.

Ultimately the ability to "live program full-stack", and to program "cyber-physically", is not simply pedagogical but instrumental. Making the full-stack available for run-time modification changes the programmer's perspective about what it means to construct software. Change becomes the norm rather than the exception, code becomes ephemeral rather than concrete, and epistemic action becomes a valid method of enquiry.

Extempore is freely available from `https://github.com/digego/extempore` under an MIT license.

References

[1] Samuel Aaron and Alan F Blackwell. "From sonic Pi to overtone: creative musical experiences with domain-specific and functional languages". In: *Proceedings of the first ACM SIGPLAN workshop on Functional art, music, modeling & design*. ACM. 2013, pp. 35–46.

[2] Harold Abelson and Gerald Jay Sussman. *Structure and interpretation of computer programs*. MIT Press, 1983.

[3] N. I. Adams IV et al. "Revised5 Report on the Algorithmic Language Scheme". In: *SIGPLAN Not.* 33.9 (Sept. 1998), pp. 26–76. ISSN: 0362-1340. DOI: 10.1145/290229.290234. URL: http://doi.acm.org/10.1145/290229.290234.

[4] Rajeev Alur. *Principles of cyber-physical systems*. MIT Press, 2015.

[5] Deb Aronson. "LLVM: The World's Compiler". In: *Click Magazine* 2 (2013), pp. 10–13.

[6] Henry G Baker. "The treadmill: real-time garbage collection without motion sickness". In: *ACM Sigplan Notices* 27.3 (1992), pp. 66–70.

[7] Karim Barkati and Pierre Jouvelot. "Synchronous programming in audio processing: A lookup table oscillator case study". In: *ACM Computing Surveys (CSUR)* 46.2 (2013), p. 24.

[8] Kent Beck et al. "Manifesto for agile software development". In: (2001).

[9] Manuel Benet. *Interview with Dennis M. Ritchie*. May 2016. URL: http://www.linuxfocus.org/English/July1999/article79.html.

[10] Jean-Philippe Bernardy et al. "A comparison of C++ concepts and Haskell type classes". In: *Proceedings of the ACM SIGPLAN workshop on Generic programming*. ACM. 2008, pp. 37–48.

[11] JM Borwein and Keith Devlin. *The computer as crucible*. AK Peters, Natick, MA, 2008.

[12] Frédéric Boussinot and Robert De Simone. "The ESTEREL language". In: *Proceedings of the IEEE* 79.9 (1991), pp. 1293–1304.

[13] Chandrasekhar Boyapati et al. "Ownership types for safe region-based memory management in real-time Java". In: *ACM SIGPLAN Notices*. Vol. 38. 5. ACM. 2003, pp. 324–337.

[14] Hank Bromley, H Bromley, and Richard Lamson. *LISP Lore: A Guide to Programming the LISP Machine Second Edition*. Springer, 1987.

[15] Rodney A. Brooks. "Intelligence without representation". In: *Artificial Intelligence* 47.1 (1991), pp. 139–159. ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(91)90053-M. URL: http://www.sciencedirect.com/science/article/pii/000437029190053M.

[16] Timothy Broomhead et al. "Virtualize Everything but Time." In: *OSDI*. Vol. 10. 2010, pp. 1–6.

[17] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. Addison-Wesley Educational Publishers Inc, 2009.

[18] Eric Cheng and Paul Hudak. *Audio processing and sound synthesis in Haskell*. Tech. rep. Technical Report YALEU/DCS/RR-1405, Yale University, 2009.

[19] Douglas J Collinge. *MOXIE: a language for computer music performance*. Ann Arbor, MI: Michigan Publishing, University of Michigan Library, 1984.

[20] Nick Collins et al. "Live coding in laptop performance". In: *Organised Sound* 8.03 (2003), pp. 321–330.

[21] Roger B Dannenberg. "The cmu midi toolkit". In: *ICMC*. Vol. 86. 1986, pp. 53–56.

[22] Viktor K Decyk. *Skeleton Codes Website UCLA*. `http://picksc.idre.ucla.edu/software/skeleton-code/`.

[23] Viktor K Decyk and Tajendra V Singh. "Particle-in-Cell algorithms for emerging computer architectures". In: *Computer Physics Communications* 185.3 (2014), pp. 708–719.

[24] Peter E Dewdney et al. "The square kilometre array". In: *Proceedings of the IEEE* 97.8 (2009), pp. 1482–1496.

[25] Paul Dourish. *Where the action is: the foundations of embodied interaction*. The MIT Press, 2004.

[26] Jonathan Edwards. "Subtext: uncovering the simplicity of programming". In: *ACM SIGPLAN Notices*. Vol. 40. 10. ACM. 2005, pp. 505–518.

[27] Daniel Frampton et al. "Demystifying magic: high-level low-level programming". In: *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. ACM. 2009, pp. 81–90.

[28] Ron Garret. *Lisping at JPL*. 2017. URL: `http://www.flownet.com/gat/jpl-lisp.html`.

[29] Jason Gregory. *Game engine architecture*. crc Press, 2009.

[30] D Griffiths. "Fluxus, 2014". In: *URL http://www.pawfal.org/fluxus* ().

[31] Jayavardhana Gubbi et al. "Internet of Things (IoT): A vision, architectural elements, and future directions". In: *Future Generation Computer Systems* 29.7 (2013), pp. 1645–1660.

[32] Jungwoo Ha et al. "A concurrent trace-based just-in-time compiler for single-threaded javascript". In: *Workshop on Parallel Execution of Sequential Programs on Multi-Core Architectures*. 2009, pp. 47–54.

[33] James Hague. *On Being Sufficiently Smart*. Mar. 2016. URL: `http://prog21.dadgum.com/40.html`.

[34] Christopher Michael Hancock. "Real-time programming and the big ideas of computational literacy". PhD thesis. Massachusetts Institute of Technology, 2003.

[35] Stefan Hanenberg. "An Experiment About Static and Dynamic Type Systems: Doubts About the Positive Impact of Static Type Systems on Development Time". In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '10. Reno/Tahoe, Nevada, USA: ACM, 2010, pp. 22–35. ISBN: 978-1-4503-0203-6. DOI: `10.1145/1869459.1869462`. URL: `http://doi.acm.org/10.1145/1869459.1869462`.

[36] Klaus Havelund et al. "Formal analysis of the remote agent before and after flight". In: *Proceedings of the 5th NASA Langley Formal Methods Workshop*. Vol. 134. 2000.

[37] Martin Heidegger. *Being and time, trans. J. Macquarrie and E. Robinson*. New York: Harper & Row, 1962.

[38] Scott Hemmert. "Green hpc: From nice to necessity". In: *Computing in Science & Engineering* 6 (2010), pp. 8–10.

[39] Daniel D Hils. "Visual languages and computing survey: Data flow visual programming languages". In: *Journal of Visual Languages & Computing* 3.1 (1992), pp. 69–101.

[40] Saurabh Hukerikar and Christian Engelmann. "Language Support for Reliable Memory Regions". In: *International Workshop on Languages and Compilers for Parallel Computing*. Springer. 2016, pp. 73–87.

[41] John Hules and Jon Bashor. *Report of the 3rd DOE Workshop on HPC Best Practices: Software Lifecycles*. Tech. rep. US Department of Energy, 2009.

[42] Dan Ingalls et al. "Back to the future: the story of Squeak, a practical Smalltalk written in itself". In: *ACM SIGPLAN Notices*. Vol. 32. 10. ACM. 1997, pp. 318–326.

[43] Stefan Kaes. "Parametric overloading in polymorphic programming languages". In: *European Symposium on Programming*. Springer. 1988, pp. 131–144.

[44] Wayne Kelly et al. "A communication model and partitioning algorithm for streaming applications for an embedded MPSoC". In: *International Symposium on System-on-Chip (SoC)*. IEEE. 2014, pp. 1–6.

[45] Glenn Krasner. *Smalltalk-80: Bits of History*. Addison-Wesley, 1983.

[46] Leslie Lamport. "Time, clocks, and the ordering of events in a distributed system". In: *Communications of the ACM* 21.7 (1978), pp. 558–565.

[47] Chris Lattner. "LLVM and Clang: Next generation compiler technology". In: *The BSD Conference*. 2008, pp. 1–2.

[48] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE. 2004, pp. 75–86.

[49] Paul Le Guernic et al. "Signal–A data flow-oriented language for signal processing". In: *IEEE transactions on acoustics, speech, and signal processing* 34.2 (1986), pp. 362–374.

[50] E.A. Lee. "Computing needs time". In: *Communications of the ACM* 52.5 (2009), pp. 70–79.

[51] Edward A Lee. "Cyber-physical systems-are computing foundations adequate". In: *Position Paper for NSF Workshop On Cyber-Physical Systems: Research Motivation, Techniques and Roadmap*. Vol. 2. Citeseer. 2006.

[52] Jie Liu and Edward A Lee. "Timed multitasking for real-time embedded software". In: *IEEE Control Systems* 23.1 (2003), pp. 65–75.

[53] T. Magnusson. "ixi lang: a SuperCollider parasite for live coding". In: *Proceedings of the International Computer Music Conference*. University of Huddersfield. 2011.

[54] Thor Magnusson. "Herding cats: Observing live coding in the wild". In: *Computer Music Journal* 38.1 (2014), pp. 8–16.

[55] Marta Mattoso et al. "Dynamic steering of HPC scientific workflows: A survey". In: *Future Generation Computer Systems* 46 (2015), pp. 100–113.

[56] Clemens Mayer et al. "An Empirical Study of the Influence of Static Type Systems on the Usability of Undocumented Software". In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '12. Tucson, Arizona, USA: ACM, 2012, pp. 683–702. ISBN: 978-1-4503-1561-6. DOI: 10.1145/2384616.2384666. URL: http://doi.acm.org/10.1145/2384616.2384666.

[57] James McCartney. "Rethinking the computer music language: SuperCollider". In: *Computer Music Journal* 26.4 (2002), pp. 61–68.

[58] Sean McDirmid. "Living it up with a live programming language". In: *ACM SIGPLAN Notices*. Vol. 42. 10. ACM. 2007, pp. 623–638.

[59] Frank McSherry, Michael Isard, and Derek G Murray. "Scalability! but at what COST?" In: *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. 2015.

[60] L.B. Meyer. *Emotion and meaning in music*. University of Chicago Press, 1956.

[61] Floréal Morandat et al. "Evaluating the design of the R language". In: *ECOOP 2012–Object-Oriented Programming* (2012), pp. 104–131.

[62] Chaker Nakhli et al. "Efficient region-based memory management for resource-limited real-time embedded systems". In: *Implementation, compilation, optimization of object-oriented languages, programs and systems (ICOOOLPS)* (2006).

[63] *NASA Deep Space 1 Website*. `https://www.jpl.nasa.gov/missions/deep-space-1-ds1/`.

[64] Allen Newell and Herbert A Simon. "Computer science as empirical inquiry: Symbols and search". In: *Communications of the ACM* 19.3 (1976), pp. 113–126.

[65] Roland Perera. "Interactive functional programming". PhD thesis. University of Birmingham, 2013.

[66] Alan J. Perlis. "Epigrams on programming". In: *SIgPLAN Notices* 17.9 (1982), pp. 7–13.

[67] Benjamin C Pierce and David N Turner. "Local type inference". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22.1 (2000), pp. 1–44.

[68] Daniel Pilaud, N Halbwachs, and JA Plaice. "LUSTRE: A declarative language for programming synchronous systems". In: *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (14th POPL 1987). ACM, New York, NY*. Vol. 178. 1987, p. 188.

[69] Richard Potter. *Just-in-time programming*. MIT Press, Cambridge, MA, 1993.

[70] Markus Rittenbruch et al. "The cube: a very large-scale interactive engagement space". In: *Proceedings of the 2013 ACM international conference on Interactive tabletops and surfaces*. ACM. 2013, pp. 1–10.

[71] Kuchera-Morin J Roberts C. "Gibber: Live Coding Audio In The Browser". In: *In Proceedings of the International Computer Music Conference (ICMC) Ljubljana, Slovenia*. ICMC. 2012.

[72] Christine Rochange, Pascal Sainrat, and Sascha Uhrig. *Time-Predictable Architectures*. John Wiley & Sons, 2014.

[73] Julian Rohrhuber, Alberto de Campo, and Renate Wieser. "Algorithms today notes on language design for just in time programming". In: *International Computer Music Conference*. 2005, p. 291.

[74] Tiark Rompf et al. "Surgical precision JIT compilers". In: *Acm Sigplan Notices*. Vol. 49. 6. ACM. 2014, pp. 41–52.

[75] Erik Sandewall. "Programming in an interactive environment: the LISP experience". In: *ACM Computing Surveys* 10.1 (1978), pp. 35–71.

[76] A. Sorensen. "A distributed memory for networked livecoding performance". In: *International Computer Music Conference. ICMA, ICMA*. 2010.

[77] A. Sorensen and H. Gardner. In: *ACM SIGPLAN Notices*. Vol. 45. 10. ACM. 2010, pp. 822–834.

[78] A.C. Sorensen. "Impromptu: An interactive programming environment for composition and performance". In: *Proceedings of the Australasian Computer Music Conference 2005*. Australasian Computer Music Association. 2005.

[79] Andrew Sorensen. *Impromptu Compiler Runtime*. Jan. 2010. URL: `http://impromptu.moso.com.au/extras/ICR.html`.

[80] Andrew Sorensen and Henry Gardner. "Systems Level Liveness with Extempore". In: *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2017. Vancouver, BC, Canada: ACM, 2017, pp. 214–228. ISBN: 978-1-4503-5530-8. DOI: `10.1145/3133850.3133858`. URL: `http://doi.acm.org/10.1145/3133850.3133858`.

[81] Andrew Sorensen, Ben Swift, and Alistair Riddell. "The Many Meanings of Live Coding". In: *Computer Music Journal* 38.1 (2014), pp. 65–76.

[82] Dimitrios Souflis. *TinyScheme*. Apr. 2016. URL: `http://tinyscheme.sourceforge.net/`.

[83] Ben Swift, Henry Gardner, and Andrew Sorensen. "Networked livecoding at VL/HCC 2013". In: *Visual Languages and Human-Centric Computing (VL/HCC), 2014 IEEE Symposium on*. IEEE. 2014, pp. 221–222.

[84] Ben Swift et al. "Coding livecoding". In: *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*. ACM. 2014, pp. 1021–1024.

[85] Ben Swift et al. "Live Programming in Scientific Simulation". In: *Supercomputing frontiers and innovations* 2.4 (2016), pp. 4–15.

[86] Ben Swift et al. "Visual code annotations for cyberphysical programming". In: *Proceedings of the 1st International Workshop on Live Programming*. IEEE Press. 2013, pp. 27–30.

[87] Steven L Tanimoto. "A perspective on the evolution of live programming". In: *Live Programming (LIVE), 2013 1st International Workshop on*. IEEE. 2013, pp. 31–34.

[88] Steven L Tanimoto. "VIVA: A visual language for image processing". In: *Journal of Visual Languages & Computing* 1.2 (1990), pp. 127–139.

[89] Mads Tofte and Jean-Pierre Talpin. "Data region inference for polymorphic functional languages". In: *Proceedings of POPL'94*. 1994, pp. 188–201.

[90] Mads Tofte and Jean-Pierre Talpin. "Region-based memory management". In: *Information and computation* 132.2 (1997), pp. 109–176.

[91] Mads Tofte et al. "A retrospective on region-based memory management". In: *Higher-Order and Symbolic Computation* 17.3 (2004), pp. 245–265.

[92] TopLap. *TopLap Manifesto*. `https://toplap.org/wiki/ManifestoDraft`. 2017.

[93] Unity. *Unity*. `http://Unity3d.com`. 2017.

[94] Philip Wadler. *The Expression Problem*. `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`. 1998.

[95] Janet H Walker et al. "The symbolics Genera programming environment". In: *IEEE Software* 4.6 (1987), p. 36.

[96] Ge Wang, Perry R Cook, et al. "ChucK: A concurrent, on-the-fly audio programming language". In: *Proceedings of the International Computer Music Conference*. Singapore: International Computer Music Association (ICMA). 2003, pp. 219–226.

[97] Stephen A Ward and Robert H Halstead. *Computation structures*. MIT press, 1990.

[98] Matthew Wright. "Open Sound Control: an enabling technology for musical networking". In: *Organised Sound* 10.03 (2005), pp. 193–200.

The Expression Problem

A

The Expression Problem Philip Wadler, 12 November 1998

The Expression Problem is a new name for an old problem. The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (e.g., no casts). For the concrete example, we take expressions as the data type, begin with one case (constants) and one function (evaluators), then add one more construct (plus) and one more function (conversion to a string).

Whether a language can solve the Expression Problem is a salient indicator of its capacity for expression. One can think of cases as rows and functions as columns in a table. In a functional language, the rows are fixed (cases in a datatype declaration) but it is easy to add new columns (functions). In an object-oriented language, the columns are fixed (methods in a class declaration) but it is easy to add new rows (subclasses). We want to make it easy to add either rows or columns.

[94]

```
;; Literal
(bind-type Lit <!a>)

(bind-func eval:[!a,Lit*]*
  (lambda (x)
    (tref x 0)))

;; Add
(bind-type Add <!a,!b>)

(bind-func eval:[!a,Add*]*
  (lambda (x)
    (+ (eval (tref x 0))
       (eval (tref x 1)))))


;; First Test add Literals
(bind-func expr_problem
  (lambda ()
    (let ((e1 (Add (Lit 1)   (Lit 2)))
          (e2 (Add (Lit 4.0) (Add (Lit 2.0) (Lit 3.0)))))
      (println "e1 =" (eval e1))
      (println "e2 =" (eval e2))
      void)))

(expr_problem)
```

Listing 68: The Expression Problem Part1

```
;; Demonstrate that we can add a new type 'sub'
;; without otherwise changing pre-existing code!

(bind-type Sub <!a,!b>)

(bind-func eval:[!a,Sub*]*
  (lambda (x)
    (- (eval (tref x 0))
       (eval (tref x 1)))))

;; Subs and Adds working together!
;; No changes required to either Lit or Add
(bind-func expr_problem
  (lambda ()
    (let ((e1 (Add (Lit 1)  (Lit 2)))
          (e2 (Sub (Lit 10) e1)))
      (println "e1 =" (eval e1))
      (println "e2 =" (eval e2))
      void)))

(expr_problem)
```

Listing 69: The Expression Problem Part2

```
;; Demonstrate that it is easy to add new operation 'stringify'
;; without otherwise changing pre-existing code!

(bind-func stringify:[!a,Lit*]*
  (lambda (x)
    (toString (tref x 0))))

(bind-func stringify:[String*,Add*]*
  (lambda (x)
    (cat (Str "(")
         (stringify (tref x 0))
         (Str " + ")
         (stringify (tref x 1))
         (Str ")"))))

(bind-func stringify:[String*,Sub*]*
  (lambda (x)
    (cat (stringify (tref x 0))
         (Str " - ")
         (stringify (tref x 1)))))

(bind-func expr_problem
  (lambda ()
    (let ((e1 (Add (Lit 1)  (Lit 2)))
          (e2 (Sub (Lit 10) e1)))
      (println (stringify e1) "=" (eval e1))
      (println (stringify e2) "=" (eval e2))
      void)))

(expr_problem)
```

Listing 70: The Expression Problem Part3

List

B

An abbreviated version of `List` taken from Extempore's core library. Extempore's core library implementation uses tail recursive definitions. Here we present non tail recursive definitions for clarity.

```
(bind-data List{!a}
           (Nil)
           (Cons !a List{!a}*))

(bind-func append:[List{!a}*,List{!a}*,List{!a}*]*
  (lambda (as bs)
    (Cons$ as (x xs) (Cons x (append xs bs)) bs)))

(bind-func join:[List{!a}*,List{List{!a}*}*]*
  (lambda (M)
    (Cons$ M (xs xss) (append xs (join xss)) (Nil))))

(bind-func fmap:[List{!b}*,[!b,!a]*,List{!a}*]*
  (lambda (f F)
    (Cons$ F (x xs) (Cons (f x) (fmap f xs)) (Nil))))

(bind-func flatmap:[List{!b}*,List{!a}*,[List{!b}*,!a]*]*
  (lambda (M f)
    (join (fmap f M))))

(bind-func unit:[List{!a}*,!a]*
  (lambda (x)
    (Cons x (Nil))))
```

Listing 71: List Library Part1

```
(bind-func apply:[List{!b}*,List{[!b,!a]*}*,List{!a}*]*
  (lambda (F1 F2)
    (Cons$ F1 (f fs)
           (Cons$ F2 (x xs)
                  (Cons (f x) (apply fs xs))
                  (Nil))
           (Nil))))

(bind-func foldl:[!b,[!b,!b,!a]*,!b,List{!a}*]*
  (lambda (fn start lst)
    (Cons$ lst (x xs)
           (foldl fn (fn start x) xs)
           start)))

(bind-func foldr:[!b,[!b,!b,!a]*,!b,List{!a}*]*
  (lambda (fn start lst)
    (Cons$ lst (x xs)
           (fn x (foldr fn start xs))
           start)))

(bind-func reverse:[List{!a}*,List{!a}*]*
  (lambda (lst)
    (Cons$ lst (x xs)
           (append (reverse xs) (unit x))
           (Nil))))
```

Listing 72: List Library Part2

Convolution Reverb

C

```
;; uniform partitioned convolution in frq domain
;; expects mono aif/wav audio file at samplerate
(bind-func creverb_c
  (lambda (filename:i8* len:i64)
    (let ((ir_len_unpadded (sf_samples_from_file filename))
          (ir_len (+ ir_len_unpadded (- len (% ir_len_unpadded len))))
          (parts (/ ir_len len)) ;; how many partitions
          (po 0)                 ;; partition offset
          (irs:SComplex* (zalloc (* 2 ir_len))) ;; storage for all IR FFT 'partitions'
          (ffts:SComplex* (zalloc (* 2 ir_len))) ;; storage for all total forward ffts
          (padlgth (* len 2)) ;; padlgth is for convolution buffers
          (aorb:i1 #t)        ;; #t for a #f for b
          (irtaila:SAMPLE* (zalloc len)) ;; buffer ir tail
          (irtailb:SAMPLE* (zalloc len))
          (drya:SAMPLE* (zalloc padlgth)) ;; buffer input signal
          (dryb:SAMPLE* (zalloc padlgth))
          (t1:i64 0) (i:i64 0) (out:SAMPLE 0.0) (j:i64 0)
          (A:SComplex* (zalloc padlgth))
          (B:SComplex* (zalloc padlgth))
          (a:SAMPLE* (zalloc padlgth))
          (t2:double 0.0)
          (output:SAMPLE* (zalloc len))) ;; buffer output
```

Listing 73: Convolution Reverb Part 1

```
;; partition and store IR spectra (circlar convolution so padded)
(dotimes (i parts)
  ;; zero pad, assumes Complexf (8 bytes per struct)
  (memset (cast A i8*) 0 (convert (* 8 padlgth)))
  (sf_read_file_into_buffer filename output (* i len) len #f)
  (Complex_bufferize output A len)
  (fft A (pref-ptr irs (* i padlgth)) padlgth))


(lambda (in:SAMPLE dry:SAMPLE wet:SAMPLE)
  (set! t2 (clock_clock))
  ;; store 'input' data which is 'ahead of out' by len samples
  (pset! (if aorb drya dryb) t1 in)
  ;; set output which is len samples behind 'in'
  (set! out ;; out is always delayed by len
        (+ (* dry (pref (if aorb dryb drya) t1)) ;; playback 'delayed' input
           (* 4.0 wet             ;; wet output signal
              (+ (pref output t1) ;; overlap-add current-output with ...
                 (pref (if aorb irtaila irtailb) t1))))) ;; delayed-irtail
  ;; increment time
  (set! t1 (+ t1 1))
```

Listing 74: Convolution Reverb Part 2

```
;; if we have buffered len samples then run convolution
(if (= t1 len)
    (let ((_fft:SComplex* (pref-ptr ffts (* po padlgth)))
          (_ir:SComplex* null))
      ;; forward FFT of incoming signal
      (fft (if aorb drya dryb) B padlgth)
      ;; store FFT to use 'now' and also for 'delayed' use
      (memcpy (cast _fft i8*) (cast B i8*) (convert (* padlgth 8)))
      ;; run convolution over all partitions
      (dotimes (i parts)
        (set! j (% (+ parts (- po i)) parts))
        (set! _fft (pref-ptr ffts (* j padlgth)))
        (set! _ir (pref-ptr irs (* i padlgth)))
        ;; '*' and '+' are overloaded for buffers of Complex{f,d}
        (Complex_multiplication_bybuf _fft _ir A padlgth)
        (Complex_addition_bybuf A B B padlgth))
      ;; after convolution is complete back to time domain!
      (ifft B a padlgth)
      (let ((scale:SAMPLE (/ 1.0 (convert (* len parts))))
            (tail (if aorb irtaila irtailb)))
        (dotimes (i len)
          (pset! output i (* (pref a i) scale)))
        (dotimes (i len)
          (pset! tail i (* (pref a (+ len i)) scale))))
      ;; reset everything for next cycle
      (set! aorb (if aorb #f #t)) ;; swap buffers
      (set! po (% (+ po 1) parts))
      (set! t1 0)))
out))))
```

Listing 75: Convolution Reverb Part 3

Web Resources

D

A set of online video resources accompany this thesis. These web resources include videos of presentations, performances and screencasts, by the author, directly pertaining to this thesis. Many of the thesis code Listings are also provided as "live" video screencasts. While the website is not a core component of the thesis, the video documentation conveys a level of "liveness" that is difficult to express in text. In particular, this "dynamic" content, will help to convey a greater sense of Extempore's "liveness" in practice. Readers are strongly encouraged to peruse these resources.

`http://extempore.moso.com.au/phd.html`